BAB III
PELAKSANAAN KERJA MAGANG

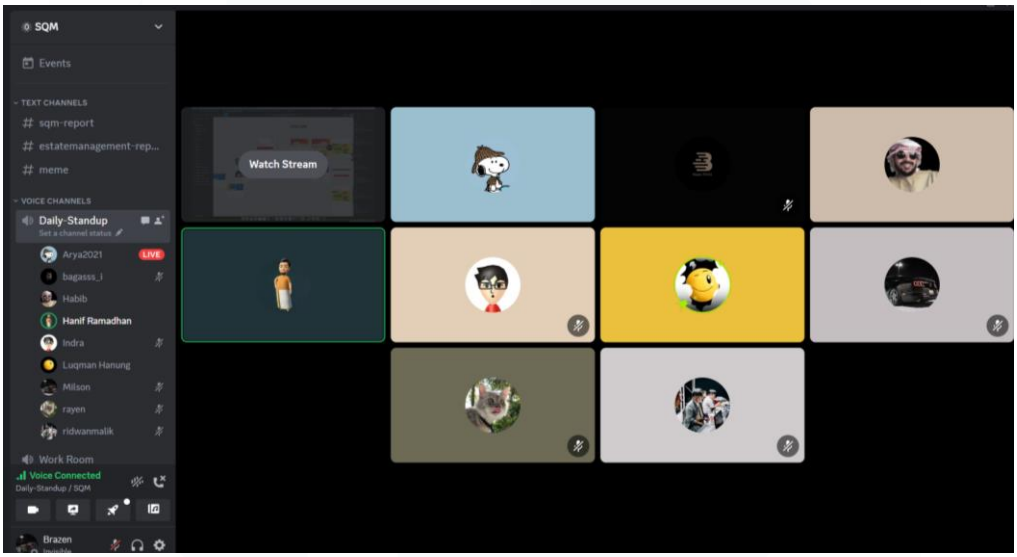
3.1 Kedudukan dan Koordinasi

Pelaksanaan kerja magang yang dilakukan di PT Quanta Land Indonesia berada pada divisi *Information Technology* sebagai seorang *Business Analyst*. Dalam menjalankan kegiatan magang, peserta dipimpin oleh Hanif Ramadhan sebagai *Team Leader* dan Shirly sebagai manajer. Hal ini dikarenakan terdapat dua tugas utama yang dilakukan, yaitu melakukan pengembangan website dan analisis bisnis pada *website* yang saat ini sedang dikembangkan. Dengan adanya analisis tersebut, perusahaan dapat mengetahui fitur atau sektor apa yang perlu dikembangkan di dalam *website*. Apabila analisis diterima, maka perubahan dan pengembangan akan dilakukan. Pengembangan tersebut akan dilakukan bersama-sama dengan melakukan perubahan penulisan pemograman dan penambahan penulisan pemograman. Selama pelaksanaan magang koordinasi dilakukan dengan menggunakan beberapa *tools* seperti Discord, Zoom, dan Talenta. Aplikasi tersebut digunakan untuk melakukan *daily meeting*, memberitahu progres terbaru pada tugas yang diberikan, dan melakukan presensi.

3.1.1 SQM *Daily Meeting*

SQM *Daily Meeting* adalah pertemuan yang wajib diikuti oleh seluruh anggota divisi IT setiap harinya pada pukul 10.00 WIB hingga 10.30 WIB melalui aplikasi Discord ataupun secara langsung apabila seluruh anggota masuk ke kantor. Kegiatan ini dilakukan untuk membahas progres setiap anggota pada tugas yang diberikan dan apa yang akan dilakukan selanjutnya oleh setiap anggotanya pada hari tersebut. Tujuan diadakan pertemuan ini juga untuk membahas dan melakukan diskusi terkait pengembangan aplikasi atau *website* serta permasalahan yang mungkin muncul. Gambar 3.1 menunjukkan *daily meeting* yang dilakukan setiap harinya melalui aplikasi *Discord*. Saat

meeting dilakukan setiap anggota bebas untuk melakukan diskusi dan menyampaikan pendapat terkait perkembangan proyek yang sedang dikerjakan.



Gambar 3.1 SQM Daily Meeting

3.2 Tugas dan Uraian Kerja Magang

Program kerja magang di PT Quanta Land Indonesia dilakukan selama 6 bulan dengan keterlibatan mahasiswa. Tugas yang dilakukan selama proses magang di PT Quanta Land Indonesia pada divisi *information technology*:

1. Membuat *flowchart* dan alur sistem pada *website* dan aplikasi SQM.
2. Melakukan analisis terhadap kompetitor SQM dan mengembangkan sistem SQM secara keseluruhan.
3. Membuat API untuk registrasi dan *login* pada *website* SQM.
4. Membuat sistem *database* SQM di MYSQL.
5. Membuat sistem *automatic assign leads* menggunakan RabbitMQ dan Redis sehingga proses terjadi secara cepat dan instan.
6. Membuat sistem *data management* untuk mengetahui peforma *sales* dan mencegah data agar tidak hilang.

Detail pelaksanaan program kerja magang di PT Quanta Land Indonesia dapat dilihat pada Tabel 3.1.

Tabel 3.1 Pelaksanaan Kerja Magang di PT Quanta Land Indonesia

No.	Deskripsi	Waktu Pengerjaan		Divisi
1.	**SQM *Flowchart and System Flow***			
a.	Eksplorasi fitur pada *website* dan aplikasi SQM	04 September 2023	05 September 2023	*Information Technology*
b.	Pembuatan *flowchart*, desain dan alur sistem *website* dan aplikasi SQM.	06 September 2023	20 September 2023	*Information Technology*
c.	Melakukan *User testing* awal untuk dibandingkan dengan kompetitor SQM.	20 September 2023	29 September 2023	*Information Technology*
2.	***Registration and Login API***			
a.	Merancang dan membuat API untuk registrasi dan *login* menggunakan JWT.	03 Oktober 2023	18 Oktober 2023	*Information Technology*
3.	**SQM A*uto Assigning Leads and Data Management***			
a.	Merancang dan membuat *prospect* API dan *assign prospect* secara manual.	16 Oktober 2023	26 Oktober 2023	*Information Technology*
b.	Merancang dan membuat *data management* pada *website.*	26 Oktober 2023	30 Oktober 2023	*Information Technology*
c.	Merancang dan membuat sistem *auto-*	30 Oktober 2023	20 November 2023	*Information Technology*

	assign leads untuk *sales* penjualan.			
d.	Membuat *unit testing* untuk mengecek kode yang telah dibuat.	21 November 2023	5 Desember 2023	*Information Technology*

3.3 Detail Pekerjaan

1. SQM *Flowchart and System Flow*

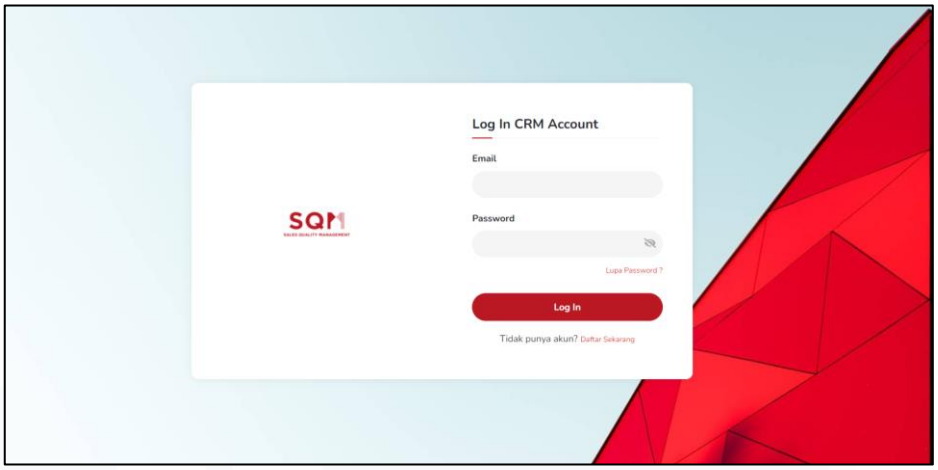
a. Eksplorasi fitur pada *website* dan aplikasi SQM

Tabel 3.2 Menu pada Website SQM

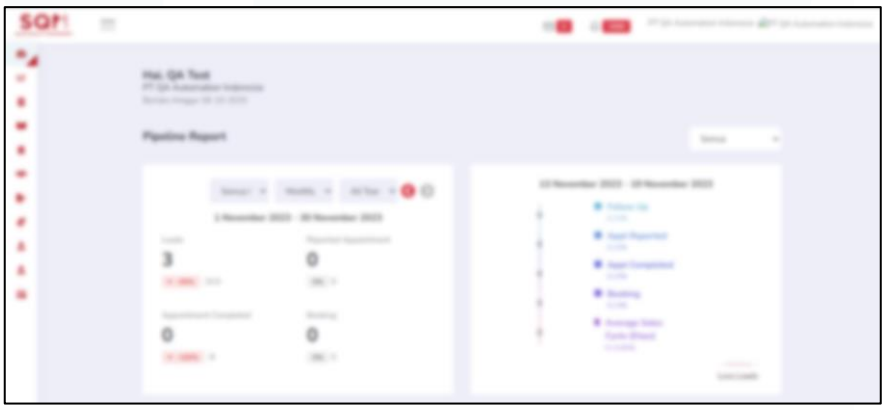
Menu	Keterangan
Dashboard	Informasi tentang *leads* yang dapat membantu penjualan
Statistik	Tempat untuk melihat performa penjualan
Listing	Informasi proyek atau properti yang dijual
Leads	Informasi terkait perkembangan *leads*
NUP	Tempat untuk melihat proses pembayaran
Booking	Tempat untuk melakukan dan mengatur aktivitas terkait pemesanan
Digital SP	Tempat pembuatan template untuk promosi pemasaran
Transaksi	Informasi tentang pembayaran bonus
My Agents	Tempat untuk melihat aktivitas agen penjualan
Member	Informasi terkait jumlah member yang terdaftar
Paket Berlangganan	Informasi terkait ketersediaan paket berlangganan dan pembayaran

Sebelum melakukan analisis pada sistem, tentunya harus dilakukan eksplorasi terlebih dahulu untuk mengerti bagaimana sistem tersebut dapat berjalan. Di dalam *website* SQM sendiri terdapat beberapa menu utama yang mendukung sistem tersebut yang dapat dilihat pada Tabel 3.2. Setiap menu memiliki fungsi dan tujuan nya masing-masing juga saling mendukung satu sama lain. Gambar 3.2 menunjukkan tampilan awal yang ada pada *website* SQM ketika masuk pertam kali ke dalam *website*.

Tabel 3.2 menunjukkan beberapa menu utama yang ada pada *website* SQM yang juga berfungsi sebagai fitur yang saling terhubung satu sama lain. Pengembangan *website* SQM akan difokuskan pada menu *Listing* dan *My Agents* dimana terdapat penugasan untuk agen penjualan dari prospek yang telah didapatkan. Di dalam menu Listing, terdapat fitur *auto-assign* yang akan menjadi fokus utama dalam pengembangan *website*. Nantinya fitur *auto-assign* akan berhubungan dengan menu *My Agents* karena di dalam menu tersebut sistem dapat melakukan pemantauan atau *monitoring* pada aktivitas agen pejualan ketika agen penjualan mengaktifkan fitur *on-duty* pada aplikasi SQM. Gambar 3.4 dan Gambar 3.6 menunjukkan tampilan pada menu *Listing* dan menu *My Agents* pada *website* SQM. Lalu, juga terdapat Gambar 3.3 yang merupakan menu *Dashboard* atau menu yang pertama kali muncul saat *User* masuk ke dalam *website* SQM seperti pada Gambar 3.2. Dalam menjalankan sistem yang ada pada *website* SQM, nantinya akan terdapat *Admin Developer* yang akan memantau para agen penjualan dan melihat progress pada prospek yang diberikan kepada agen penjualan pada menu *Leads* seperti pada Gambar 3.5.

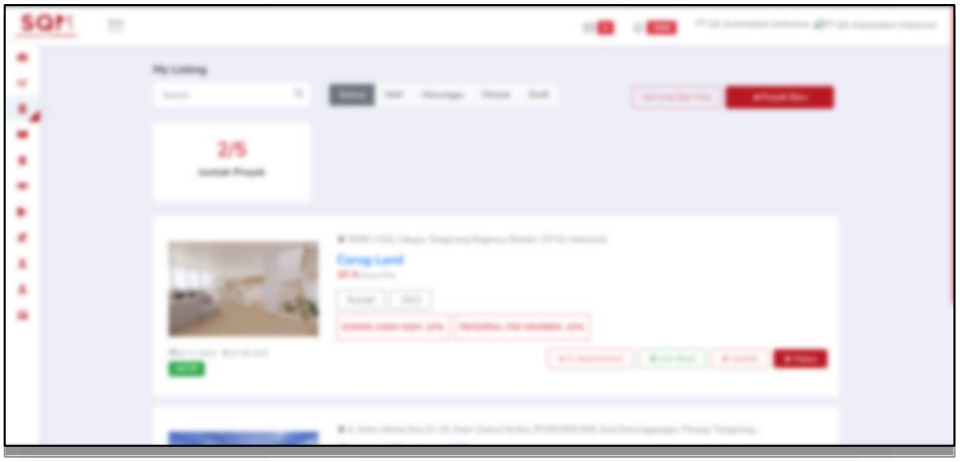


Gambar 3.2 Landing Page SQM



Gambar 3.3 Menu *Dashboard* SQM

Gambar 3.3 merupakan tampilan pada menu *Dashboard* pada *website* SQM. Menu ini berfungsi sebagai tempat untuk melihat *progress* dari prospek yang ada dan melakukan analisis serta prediksi terkait prospek yang mungkin didapatkan. Untuk melihat data prospek lebih dalam lagi, *user* dapat mengatur *timeline* yang telah disediakan. Adanya menu tentunya membuat *user* mengetahui prospek mana yang harus di *follow-up* dan jumlah prospek yang telah *user* kerjakan. Apabila *user* ingin melihat lebih detail lagi terkait peforma terkait dengan prospek, maka *user* dapat memilih opsi *Show more* yang ada pada menu *Dashboard* dan *user* akan secara otomatis berpindah halaman ke menu *Analytics*.

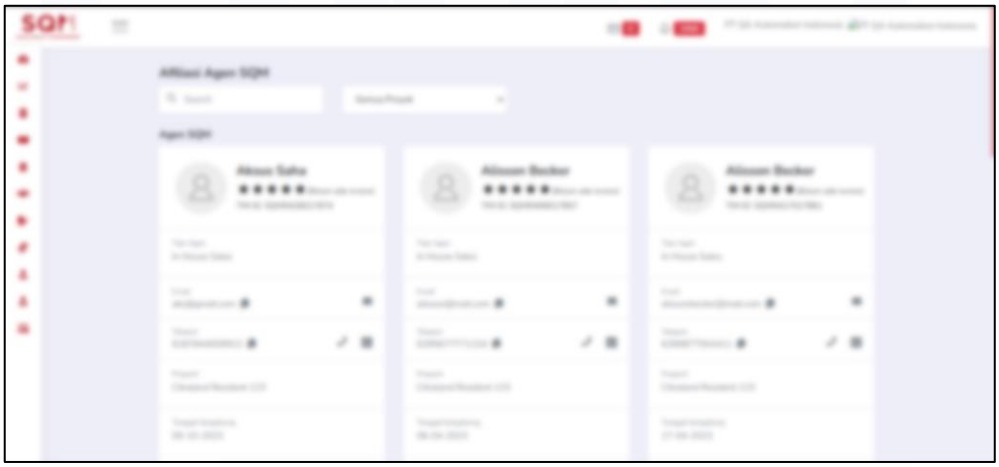


Gambar 3.4 Menu *Listing* SQM

Gambar 3.4 merupakan tampilan menu *Listing* pada *website* SQM yang berisi tentang proyek-proyek yang sedang dikerjakan oleh perusahaan ataupun proyek yang saat ini ingin dijual. Menu *Listing* memiliki banyak fungsi yang dapat membantu *user* seperti membuat jadwal *booking* bersama konsumen yang tertarik, melihat *siteplan* terkait proyek yang ada, dan memberikan informasi secara lengkap pada proyek yang dipilih oleh *user*. Menu ini sangatlah berguna dan membantu *user* dalam penjualan karena banyaknya informasi yang dapat diakses melalui menu ini.

Gambar 3.5 Menu *Leads* SQM

Gambar 3.5 merupakan tampilan menu *Leads* yang ada pada *website* SQM yang berfungsi sebagai pengatur prospek yang akan dikerjakan oleh *user*. Dalam hal ini *Sales Manager* dapat melihat dan memantau para *user* dalam melakukan pekerjaan. Di dalam menu ini juga, seluruh prospek tersimpan dari seluruh proyek yang ada. Adanya menu ini akan mempermudah *manager* dalam membagi prospek sehingga prospek dapat dipastikan sudah di*follow-up* oleh para *user*. Menu ini memiliki beberapa *submenu* untuk melakukan filter terkait kategori prospek yang ada.



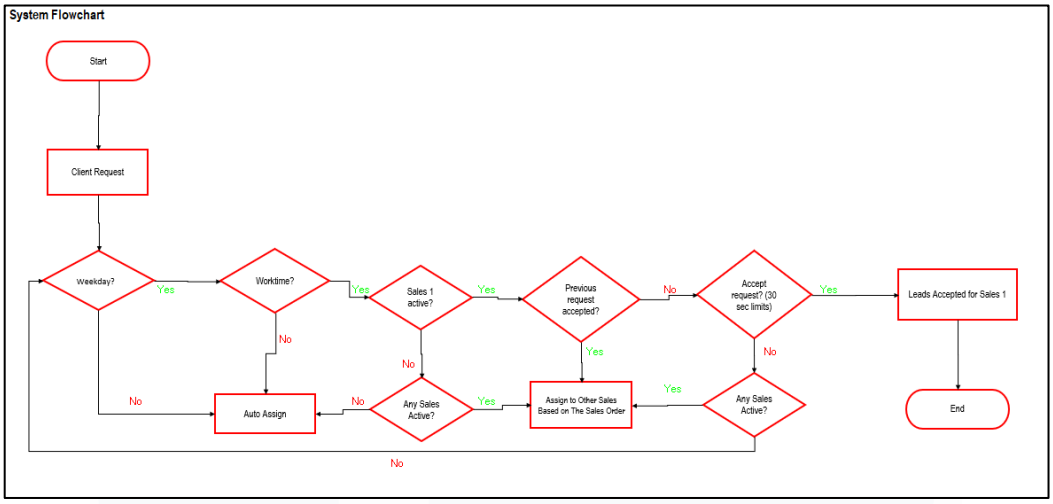
Gambar 3.6 Menu *MyAgents* SQM

Gambar 3.6 merupakan tampilan menu *MyAgents* pada *website* SQM yang berisi data terkait para agen penjualan. Di dalam menu ini dapat melihat status dari para *user* apakah sedang aktif atau tidak. Menu ini juga memiliki fitur untuk mengetahui posisi *user* yang ada dengan menggunakan fitur *Live Location*. Apabila dari konsumen atau pelanggan membutuhkan agen untuk menjelaskan informasi terkait proyek yang ada, maka menu ini akan sangat membantu karena seluruh data dari agen yang ada tercatat di menu ini.

b. Pembuatan *flowchart,* desain dan alur sistem *website* dan aplikasi SQM

1. *Flowchart* SQM

Setelah melakukan eksplorasi pada *website* dan aplikasi SQM, maka kegiatan selanjutnya adalah dengan membuat *flowchart* dan alur sistem pada aplikasi SQM. Pembuatan *flowchart* dilakukan dengan pembuatan *system flowchart* terlebih dahulu seperti pada Gambar 3.7 yang nantinya dilanjutkan dengan pembuatan *flowchart* pada masing-masing menu bedasarkan dengan ketentuan perusahaan. Terdapat dua gambar yang menunjukkan *flowchart* untuk menu *Listing* pada Gambar 3.8 dan menu *MyAgents* pada Gambar 3.9. Pembuatan *flowchart* dilakukan sesuai dengan alur kerja pada perusahaan sehingga di dalam *flowchart* juga terdapat divisi lain seperti divisi keuangan.



Gambar 3.7 System Flowchart SQM

Gambar 3.7 meruapakan tampilan *system flowchart* yang ada pada *website* SQM yang menjelaskan proses bagaimana sistem mencatat prospek yang ada dan melakukan *auto-assign* kepada prospek. Konsep ini dimulai dengan permintaan dari pelanggan, lalu penentuan pada hari kerja atau bukan untuk menentukan arah sistem nantinya. Apabila pada hari weekend maka prospek akan secara otomatis terkirim pada User sedangkan apabila pada hari kerja maka sistem akan menuju ke *auto-assign* ketika *User* sedang aktif dan meminta respon dari *User* terkait propsek yang ada. Apabila *User* menerima prospek tersebut maka *User* bertanggung jawab pada prospek yang telah diterimanya. Apabila *User* menolak prospek, maka prospek akan diberikan kepada *User* lain.

Gambar 3.8 *Flowchart Listing* SQM

Gambar 3.8 merupakan *flowchart* pada menu *Listing website* SQM yang menjelaskan proses bagaimana proyek dapat tercatat pada menu *Listing*. Dalam hal ini proyek yang akan dimasukkan harus disetujui terlebih dahulu sebelum dicantumkan ke dalam *website* yang nantinya dapat digunakan sebagai sumber informasi.

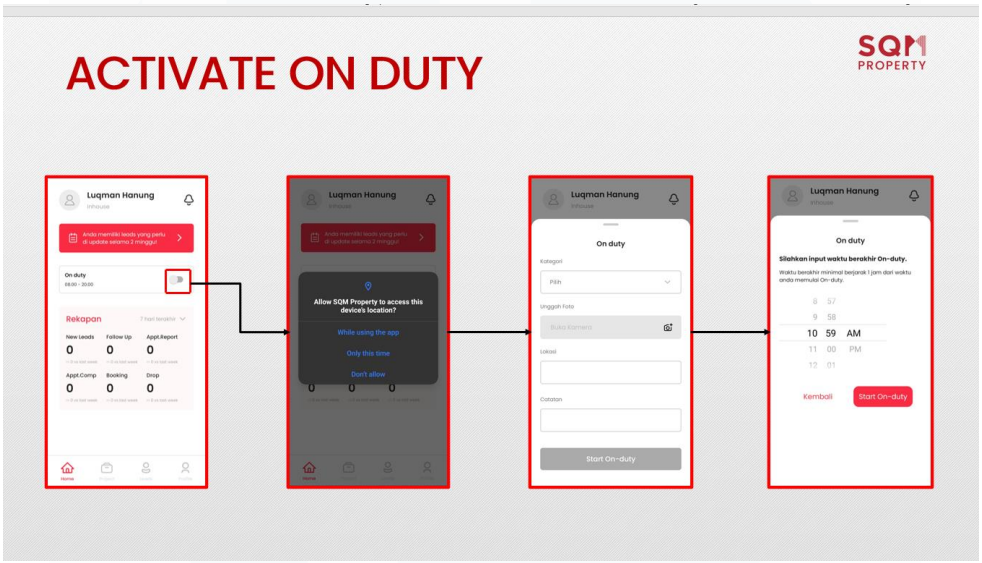
Gambar 3.9 *Flowchart MyAgents* SQM

Gambar 3.9 merupakan *flowchart* pada menu *MyAgents website* SQM yang menjelaskan bagaimana data para agen penjualan dapat tercatat dalam menu *MyAgents*. Dalam hal ini, para agen penjualan akan dipastikan apakah masih aktif bekerja atau tidak. Apabila sudah dipastikan, maka data para agen dapat dimasukkan ke dalam *website*.

2. Desain dan Alur Sistem Aplikasi SQM

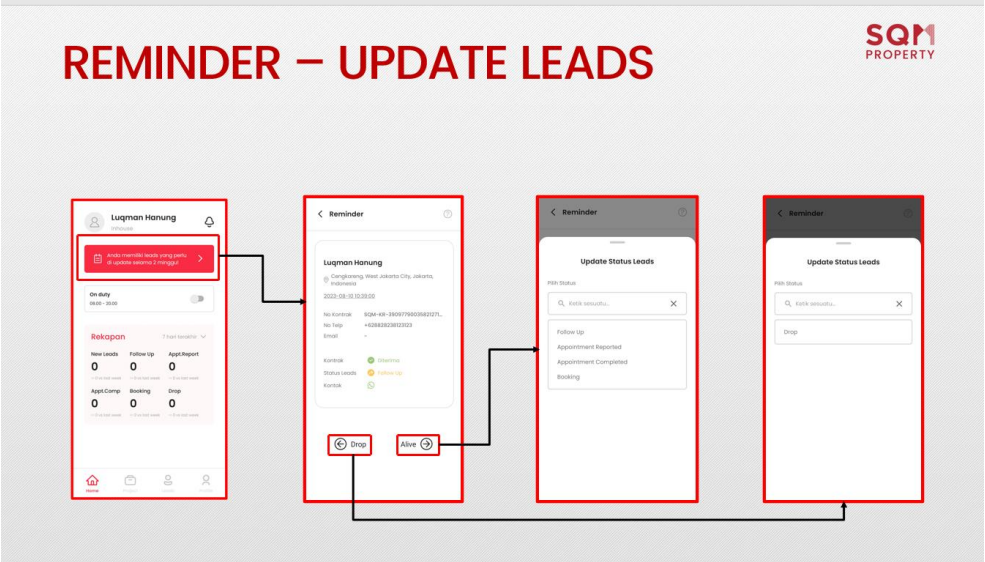
Proses pengembangan pada SQM tidak hanya dilakukan pada *website* SQM saja, tetapi juga aplikasi *mobile* SQM. Tujuan dari perubahan desain pada aplikasi SQM dikarenakan desain pada aplikasi saat ini masih kurang baik begitu pula dengan sistem aplikasi yang ada pada aplikasi SQM. Terdapat beberapa gambar yang menunjukkan desain pada aplikasi SQM saat ini. Gambar 3.10 merupakan tampilan desain pada menu utama aplikasi SQM, Gambar 3.11 merupakan tampilan desain pada menu *Leads* aplikasi SQM, dan Gambar 3.12 merupakan tampilan desain pada menu *analytics* aplikasi SQM. Perubahan yang ada pada aplikasi SQM tidak hanya berupa desain aplikasi saja, tetapi juga alur kerja dari aplikasi SQM

sehingga proses yang ada pada aplikasi dapat berjalan dengan lebih baik dan lebih efisien lagi.



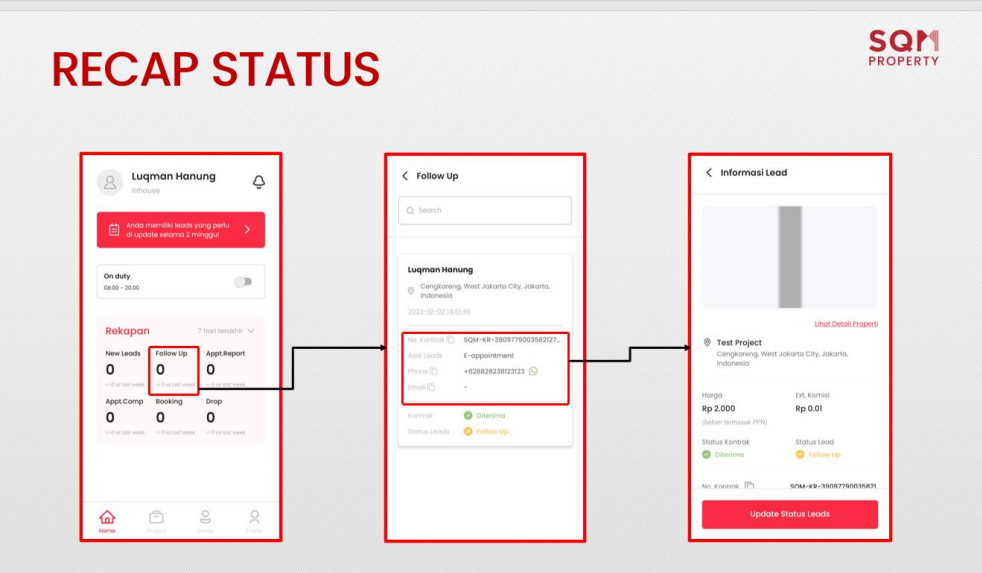
Gambar 3.10 Tampilan Desain Menu Utama pada Aplikasi SQM

Gambar 3.10 merupakan tampilan desain menu utama pada aplikasi SQM sebelum diperbarui yang berfokus pada data dari prospek yang ada saat ini bedasarkan status prospek itu sendiri. Di dalam aplikasi ini para agen penjualan juga dapat mengaktifkan fitur *on-duty* yang berarti para agen sudah siap untuk menerima prospek yang dikirimkan oleh sistem untuk dikerjakan.



Gambar 3.11 Tampilan Desain Menu *Leads* pada Aplikasi SQM

Gambar 3.11 merupakan tampilan desain menu *Leads* pada aplikasi SQM sebelum diperbarui yang berfungsi sebagai tempat untuk melakukan *update* pada prospek yang sedang dikerjakan. Menu ini juga berfungsi sebagai tempat untuk menyimpan data prospek baru yang nantinya akan terhubung dengan *website* SQM.



Gambar 3.12 Tampilan Menu *Analytics* pada Aplikasi SQM

Gambar 3.12 merupakan tampilan desain menu *Analytics* pada aplikasi SQM sebelum diperbarui yang berfungsi sebagai tempat untuk melihat analisis yang dihasilkan oleh sistem secara otomatis. Analisis yang dilakukan biasanya berhubungan dengan status kategori pada prospek yang dipilih oleh *User*.

Perubahan yang dilakukan pada desain aplikasi SQM dilakukan bersamaan dengan divisi UI/UX dimana divisi UI/UX lebih fokus dalam pembuatan desain aplikasi daripada alur kerja sistem. Setelah desain pada aplikasi SQM telah selesai, selanjutnya dilakukan *testing* terlebih dahulu di dalam aplikasi *Figma* untuk memastikan bahwa alur sistem telah sesuai. Terdapat beberapa gambar yang menunjukkan perubahan pada aplikasi SQM seperti pada Gambar 3.13 pada menu utama, Gambar 3.14 pada menu *Leads*, Gambar 3.15 pada menu *Analytics*, dan Gambar 3.16 yang merupakan tambahan fitur baru pada aplikasi SQM berupa *QR scan*.

Gambar 3.13 Tampilan Desain Baru Menu Utama pada Aplikasi SQM

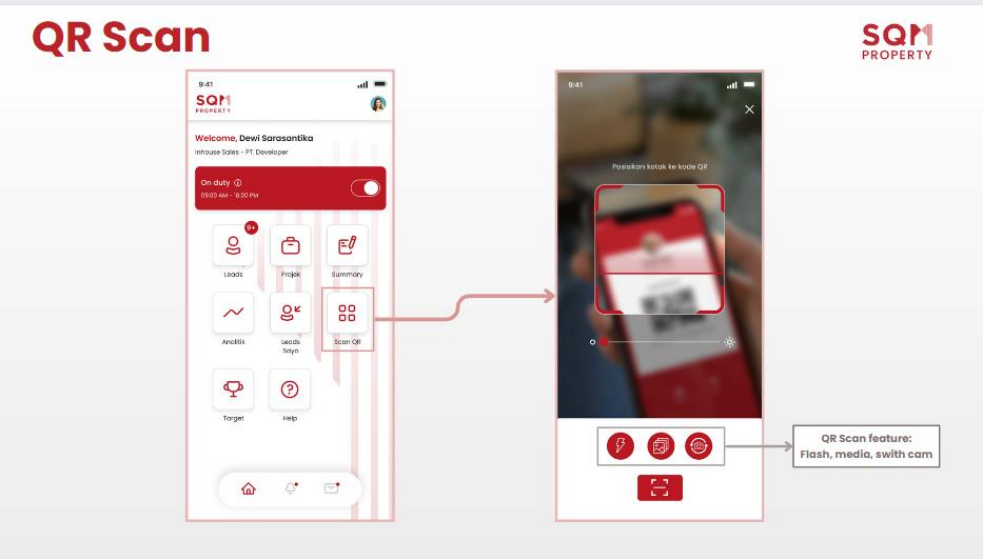
Gambar 3.13 merupakan tampilan desain baru menu utama pada aplikasi SQM. Perubahan utama yang dilakukan adalah tata letak menu dan pilihan menu yang diberikan lebih banyak sehingga mempermudah *user*. Tujuan adanya menu yang lebih banyak adalah untuk mempermudah *user* untuk menggunakan aplikasi SQM.

Gambar 3.14 Tampilan Desain Baru Menu *Leads* pada Aplikasi SQM

Gambar 3.14 merupakan tampilan desain baru menu *Leads* pada aplikasi SQM. Perubahan utama yang dilakukan adalah tampilan menu dan tampilan prospek yang dibuat berurutan sesuai dengan status yang membutuhkan *update* tercepat.

Gambar 3.15 Tampilan Desain Baru Menu *Analytics* pada Aplikasi SQM

Gambar 3.15 merupakan tampilan desain baru menu *Analytics* pada aplikasi SQM. Perubahan utama yang dilakukan adalah tata letak menu dan informasi analisis yang diberikan lebih banyak sehingga mempermudah *user* dalam mengetahui apa yang harus dilakukan nantinya. Tujuan adanya menu *Analytics* adalah membantu untuk meningkatkan penjualan.



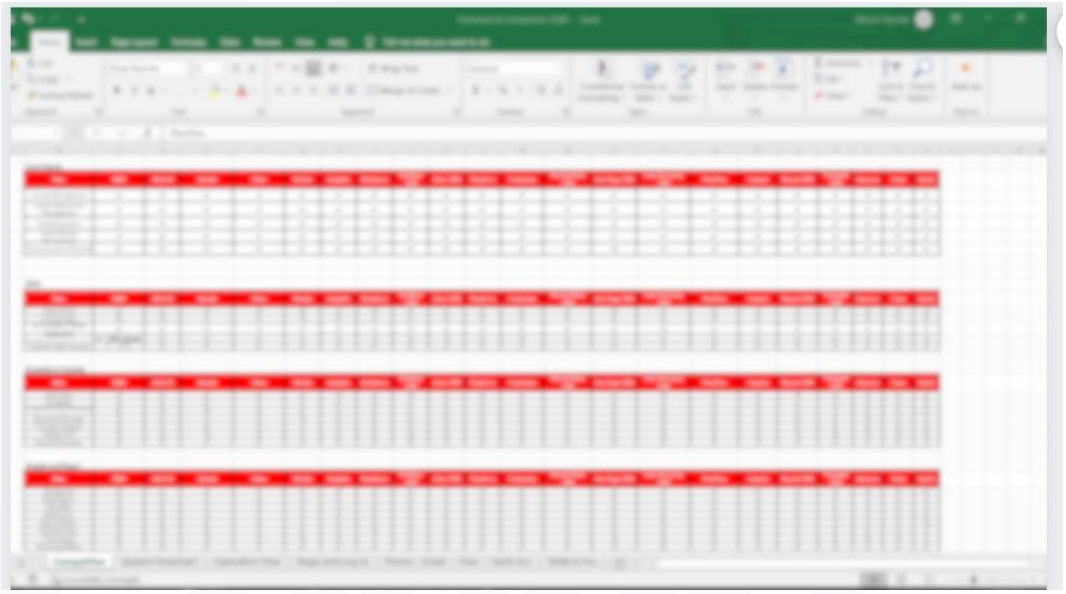
Gambar 3.16 Tampilan Desain pada Fitur Baru SQM

Gambar 3.16 merupakan tampilan desain baru menu QR pada aplikasi SQM. Penambahan fitur ini dilakukan bedasarkan kebutuhan yang ditemukan di lapangan.

c. Melakukan *User testing* awal untuk dibandingkan dengan kompetitor SQM

Sebelum mengembangkan sistem yang ada pada *website* Sales Quality Management (SQM), dilakukan *User testing* awal terlebih dahulu untuk mengenal lebih dalam terkait fitur dan sistem yang telah berjalan sebelumnya. Tujuan dari *User testing* adalah untuk mengetahui apakah fitur

dan sistem yang ada telah berjalan sesuai dengan kebutuhan pengguna. Dengan mengetahui kelebihan dan kekurangan pada sistem, maka hal ini yang akan menjadi acuan untuk mengembangkan *website*. Dengan mencari para kompetitor dan membandingkan sistem dengan sistem yang lain, maka diharapkan perkembangan *website* nantinya dapat berjalan dengan baik. Gambar 3.17 menunjukkan tabel hasil analisis yang berisi perbandingan antara *website* maupun aplikasi yang telah dianalisis untuk membantu perkembangan *website* SQM menjadi lebih baik lagi.



Gambar 3.17 Hasil Analisis Kompetitor SQM

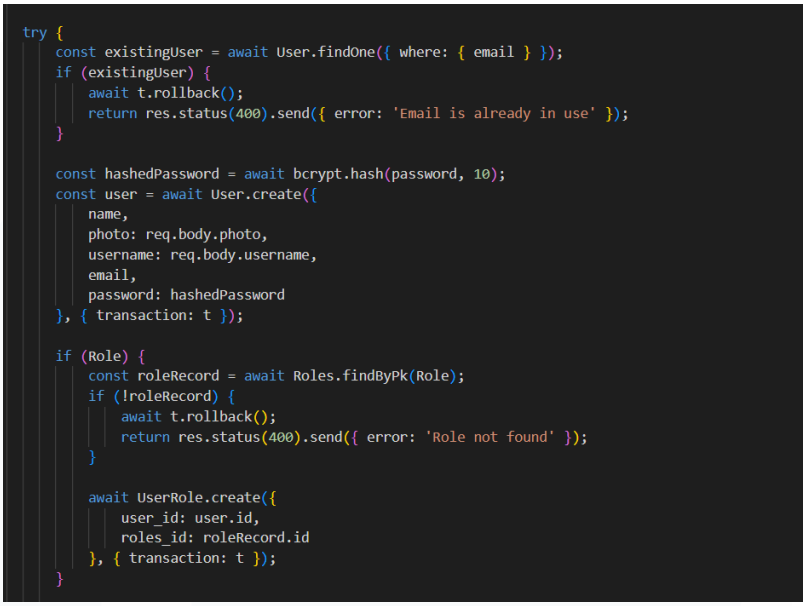
2. *Registration* dan *Login* API

a. Merancang dan membuat API untuk registrasi dan *login* menggunakan JWT.

Pembuatan API untuk *registration* dan *login* menjadi langkah awal perusahaan dalam mengembangkan sistem *website* dan aplikasi SQM. API ini digunakan untuk mendaftarkan para agen penjualan dan peran dari agen itu sendiri dalam suatu proyek yang sedang dikerjakan. Peran dari agen

terbagi menjadi enam peran, yaitu *Super Administrator, Admin Developer, General Manager, Sales Manager, Sales Agent,* dan *Staff Admin*. Setiap peran memiliki fungsinya masing-masing yang bertujuan untuk meningkatkan efisiensi dalam melakukan pekerjaan.

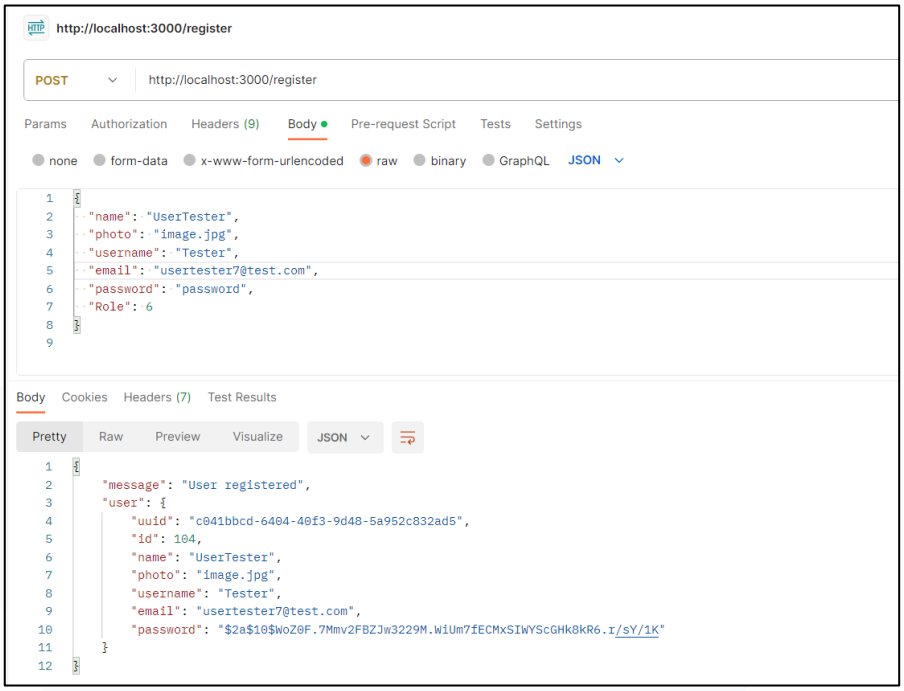
Data yang perlu diisi pada saat melakukan pendaftaran adalah nama, foto, *Username,* e-mail, *password,* dan *roles* dari *User* itu sendiri. Demi menjaga kemanan data yang tersimpan, maka setiap *Userid* pada masing-masing *User* dilakukan oleh *library uuidv4* yang secara otomatis terbuat saat adanya pendaftaran. Tak hanya itu, hal ini juga dilakukan pada *password User* yang secara otomatis akan ter-*encrypt* sehingga sulit untuk diakses bahkan dari *User* yang memiliki akses untuk melihat daftar *User* yang ada. Untuk mencegah adanya redudansi dan duplikasi data, maka dalam penerapan *registration* API menggunakan beberapa ketentuan. Ketentuan tersebut berupa *User* tidak boleh memiliki e-mail yang sama dan batas pemilihan peran juga hanya enam pilihan. Apabila ketentuan tersebut dilanggar maka sistem akan secara otomatis melakukan pembatalan pada *User* yang sedang melakukan pendaftaran akun.



```
try {
    const existingUser = await User.findOne({ where: { email } });
    if (existingUser) {
        await t.rollback();
        return res.status(400).send({ error: 'Email is already in use' });
    }

    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({
        name,
        photo: req.body.photo,
        username: req.body.username,
        email,
        password: hashedPassword
    }, { transaction: t });

    if (Role) {
        const roleRecord = await Roles.findByPk(Role);
        if (!roleRecord) {
            await t.rollback();
            return res.status(400).send({ error: 'Role not found' });
        }

        await UserRole.create({
            user_id: user.id,
            roles_id: roleRecord.id
        }, { transaction: t });
    }
```

Gambar 3.18 Kode *Registration* API

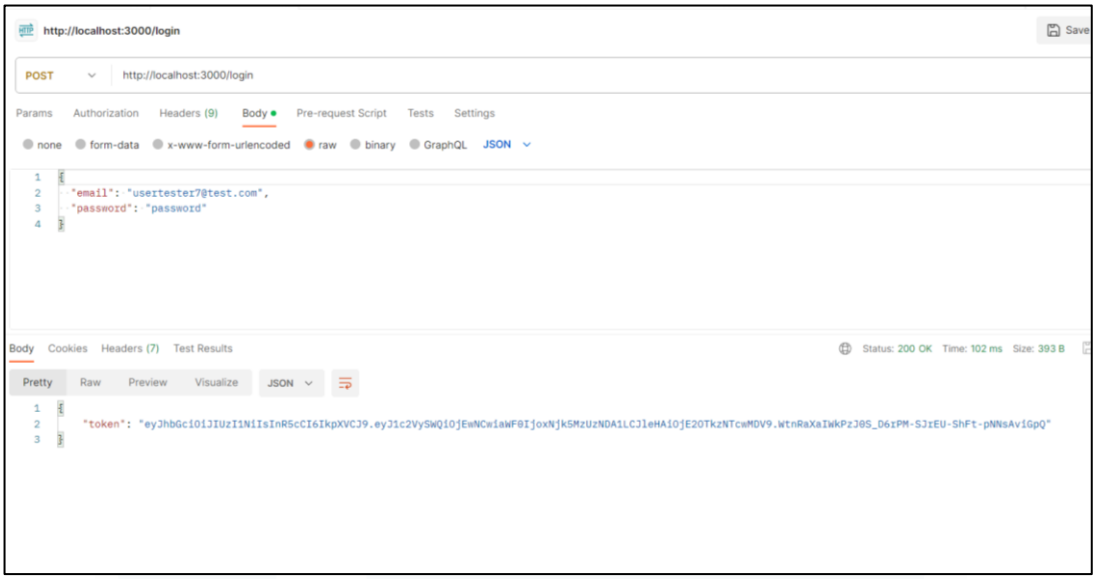
Gambar 3.18 merupakan susunan kode pada *registration* API yang melakukan pengecekan awal pada e-mail apakah terdapat duplikasi atau tidak. *User akan* gagal dalam melakukan pendaftaran karena menggunakan e-mail yang sama. Setelah melakukan pengecekan email maka data *User* akan dicatat terlebih dahulu, namun jika peran yang dipilih *User* tidak tersedia maka data *User* tidak akan terdaftar karena menggunakan sistem *rollback*, dimana apabila terjadi kesalahan maka sistem secara otomatis akan melakukan pembatalan. *User* juga akan gagal dalam melakukan pendaftaran apabila peran yang dipilih tidak ada di dalam sistem. Apabila seluruh syarat dan ketentuan telah terpenuhi maka sistem akan menerima data tersebut dan menyimpannya ke dalam *database* seperti Gambar 3.19 yang menunjukkan *User* berhasil terdaftar.



Gambar 3.19 *User* Berhasil Melakukan Registrasi di Postman

```
app.post('/login', async (req, res) => {
    try {
        const { email, password } = req.body;

        if (!email || !password) {
            return res.status(400).json({ error: 'Email and password are required' });
        }

        const user = await User.findOne({ where: { email } });

        if (!user) {
            return res.status(401).json({ error: 'Invalid email' });
        }

        const isPasswordValid = await bcrypt.compare(password, user.password);

        if (!isPasswordValid) {
            return res.status(401).json({ error: 'Invalid password' });
        }

        const token = jwt.sign({ userId: user.id }, SECRET_KEY, { expiresIn: '1h' });
```

Gambar 3.20 Kode pada Login API



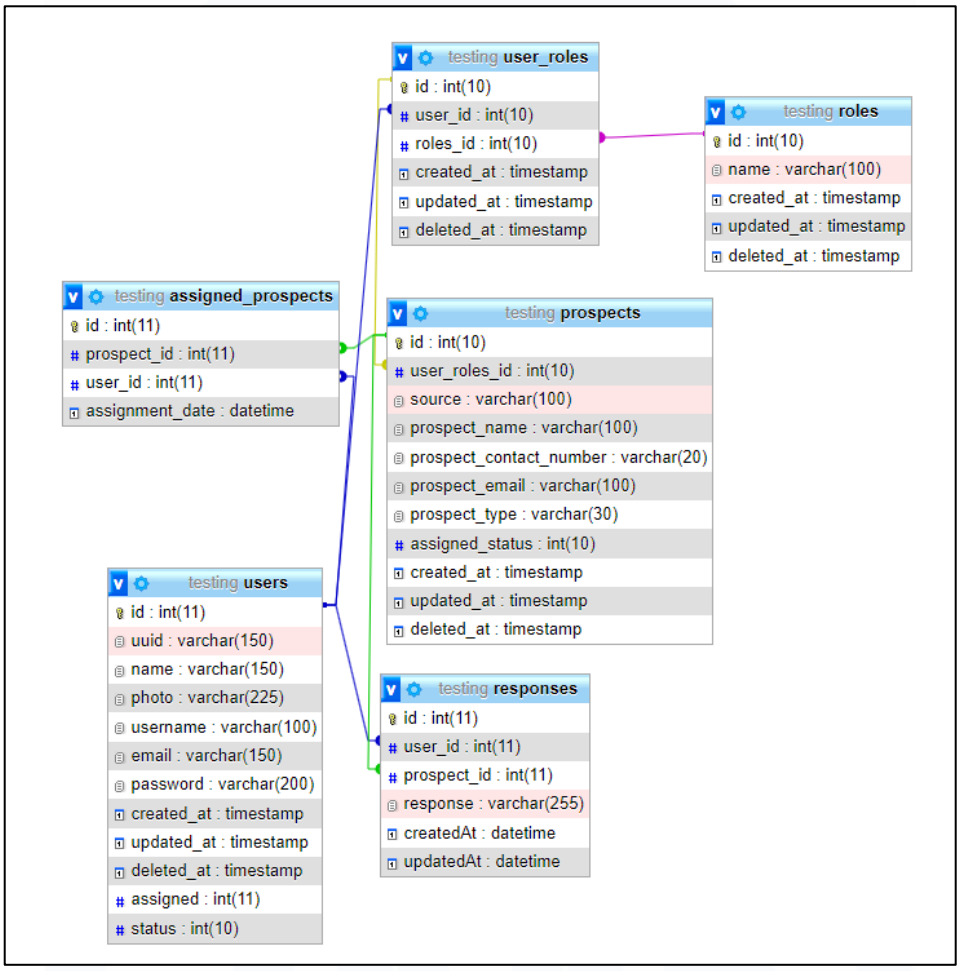
Gambar 3.21 *User* Berhasil Melakukan Login di Postman

Gambar 3.21 merupakan *User* yang berhasil melakukan *login* pada sistem yang menerima token JWT (JSON Web Token) yang hanya berlaku untuk 1 jam ke depan. Token ini dapat digunakan oleh *User* untuk masuk ke dalam sistem dan mengakses data yang ada pada sistem. Jika *User* telah melewati 1 jam, maka *User* harus melakukan *login* ulang untuk dapat mengakses data lagi. Gambar 3.20 menunjukkan kode yang mengatur batas waktu bagi *User* untuk dapat mengakses data.

3. SQM A*uto Assigning Leads and Data Management*

Pembuatan API untuk *auto-assign* dalam pengerjaannya memerlukan beberapa *tool* seperti node.js sebagai dasar logika dalam pembuatan API, MYSQL sebagai penyimpan data ke dalam *database*, RabbitMQ sebagai alat untuk mengurutkan prospek yang ada, dan Redis sebagai alat untuk melakukan *caching* data sehingga proses yang terjadi dapat dilakukan secara cepat dan efisien. Gambar 3.23 menunjukkan RabbitMQ yang sedang berjalan dan mengurutkan data dalam bentuk *message*, yang nantinya akan diurutkan melalui Redis. Gambar 3.24

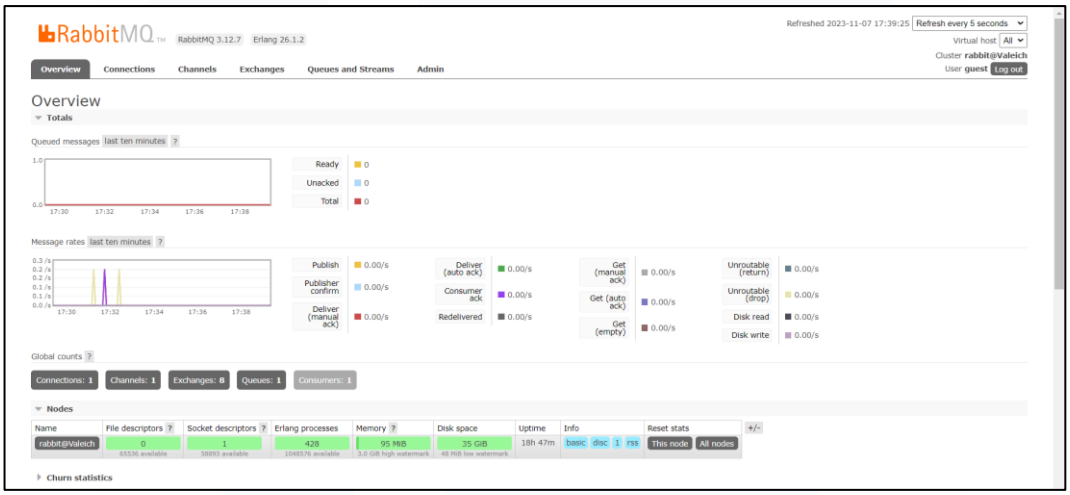
menunjukkan Redis yang siap untuk melakukan *caching* data terbukti dengan respon dari Redis yang siap untuk menerima koneksi.



Gambar 3.22 Skema Database pada Fitur *Auto-assign*

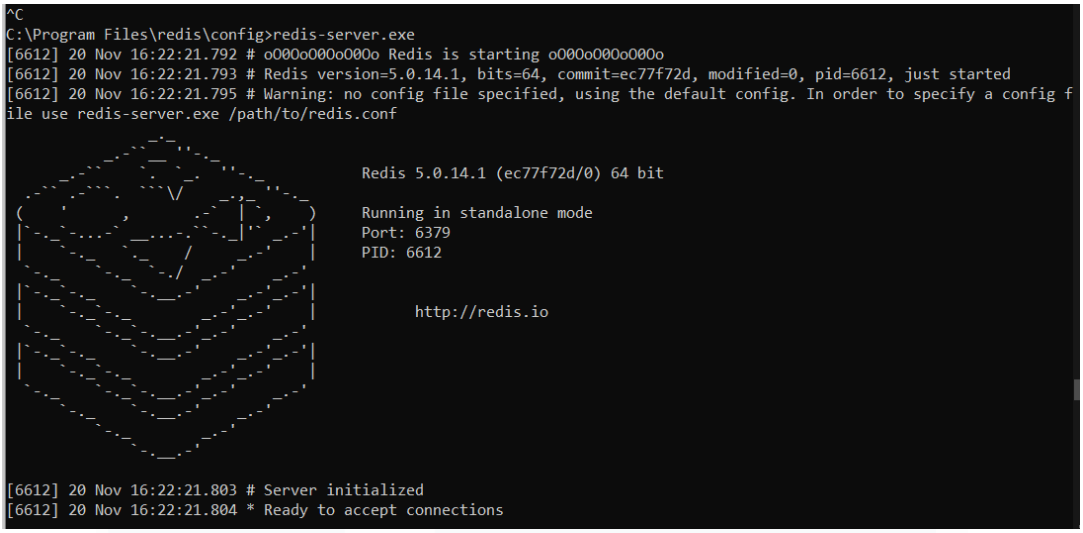
Gambar 3.22 merupakan tampilan skema *database* yang akan diterapkan nantinya ke dalam sistem utama. Dalam pembuatan sistem *auto-assign* terdapat 6 tabel utama yang digunakan dan terhubung satu sama lain. Keenam tabel ini juga secara tidak langsung terhubung satu sama lain karena masih berhubungan dengan prospek dan *user* yang mengerjakan prospek tersebut. Hal lain yang juga tak kalah pentingnya dalam skema *database* ini adalah adanya pencatatan waktu pada setiap aktivitas yang terjadi pada *database* tersebut. Aktivitas tersebut dapat berupa *user* yang

melakukan registrasi, pembuatan *roles* baru, adanya prospek baru yang masuk, dan pencatatan ketika *user* menerima sebuah prospek untuk dikerjakan.



Gambar 3.23 *Queueing Data* Menggunakan RabbitMQ

Sebelum menggunakan RabbitMQ untuk mengurutkan data pada sistem *auto-assign* nantinya, hal yang perlu diperhatikan adalah konsep dari RabbitMQ. RabbitMQ sendiri adalah sebuah penyimpanan data berbasis memori yang cepat dan dapat digunakan secara fleksibel yang berarti dapat menyimpan berbagai jenis struktur data. RabbitMQ sendiri menggunakan konsep untuk menyimpan data di dalam RabbitMQ berbentuk pesanan (*message*). Penggunaan RabbitMQ pada sistem ini bertujuan untuk mengurutkan data yang ada berupa prospek yang nantinya akan diurutkan oleh Redis. Tujuan dari penggunaan konsep ini adalah untuk meningkatkan skalabilitas dan keamanan pada data. Hal yang perlu dilakukan sebelum menggunakan RabbitMQ adalah dengan melakukan instalasi terlebih dahulu dan melakukan koneksi pada RabbitMQ terlebih dahulu. Setelah koneksi telah dilakukan maka penulisan kode dapat dilakukan.



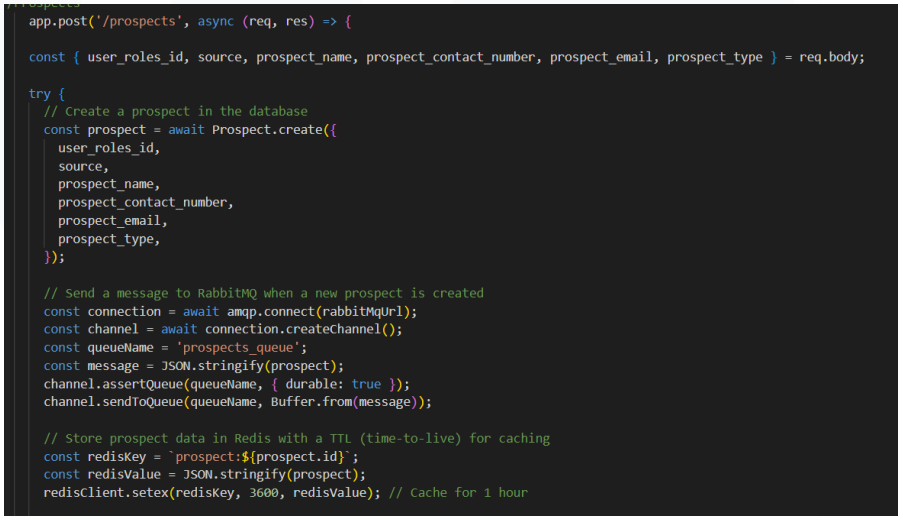
Gambar 3.24 *Caching Data* Menggunakan Redis

Redis adalah penyimpanan data berbasis memori yang prosesnya sangatlah cepat dan dapat digunakan untuk meyimpan berbagai jenis struktur data. Hal ini dikarenakan Redis menggunakan RAM untuk memproses data seperti baca dan tulis. Adanya Redis pada sistem *auto-assign* akan membantu sistem untuk mengirimkan prospek secara cepat kepada *User* sehingga prospek dapat segera ditindak lanjuti oleh *User*. Sama seperti dengan RabbitMQ, hal yang diperelukan untuk menggunakan Redis adalah dengan melakukan instalasi terlebih dahulu dan melakukan koneksi dengan sistem. Setelah koneksi telah dibuat, maka penulisan kode dapat langsung dilakukan.

a. Merancang dan membuat *prospect* API dan *assign prospect* secara manual.

Dalam menjalankan proses bisnis khususnya di bidang *real estate*, prospek menjadi peranan penting dalam mencari pelanggan dikarenakan prospek merupakan peluang dalam menemukan pembeli. Oleh karena itu, perusahaan membutuhkan sebuah sistem yang dapat mencatat dan menyimpan prospek sehingga prospek yang telah dicatat tidak hilang dan dapat digunakan lagi untuk menawarkan produk lain. Saat ini perusahaan

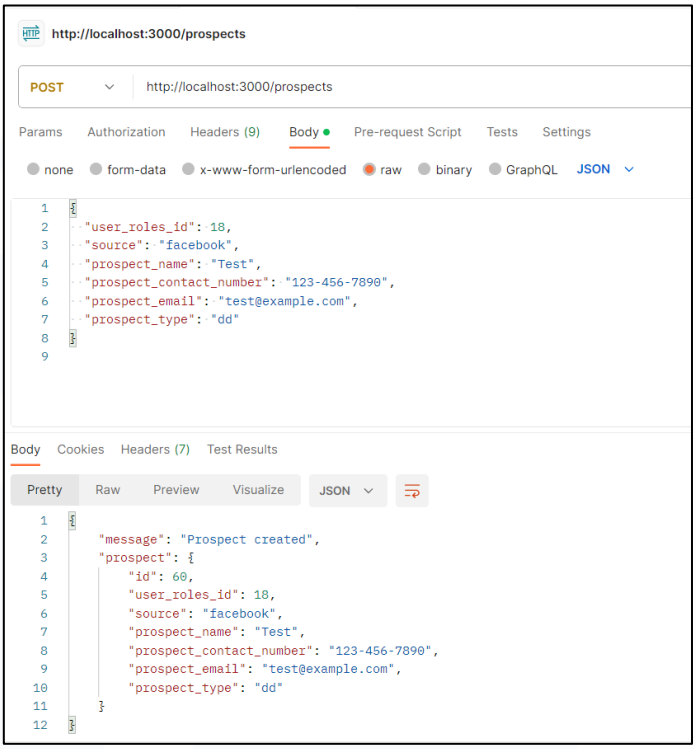
masih melakukan pencatatan prospek secara manual sehingga data prospek yang ada sering hilang dan tidak valid yang merugikan perusahaan. Untuk mempermudah agen penjualan mana yang aktif, sistem akan mencatat *User id* agen penjualan yang memberikan prospek tersebut. *Prospect* API meminta *User* untuk mengisi beberapa data seperti *source, prospect_name, prospect_contact_number, prospect_email,* dan *prospect_type.*



```
app.post('/prospects', async (req, res) => {

const { user_roles_id, source, prospect_name, prospect_contact_number, prospect_email, prospect_type } = req.body;

try {
  // Create a prospect in the database
  const prospect = await Prospect.create({
    user_roles_id,
    source,
    prospect_name,
    prospect_contact_number,
    prospect_email,
    prospect_type,
  });

  // Send a message to RabbitMQ when a new prospect is created
  const connection = await amqp.connect(rabbitMqUrl);
  const channel = await connection.createChannel();
  const queueName = 'prospects_queue';
  const message = JSON.stringify(prospect);
  channel.assertQueue(queueName, { durable: true });
  channel.sendToQueue(queueName, Buffer.from(message));

  // Store prospect data in Redis with a TTL (time-to-live) for caching
  const redisKey = `prospect:${prospect.id}`;
  const redisValue = JSON.stringify(prospect);
  redisClient.setex(redisKey, 3600, redisValue); // Cache for 1 hour
```

Gambar 3.25 Kode pada Prospect API

Gambar 3.25 merupakan kode yang dibuat dalam membuat *prospect* API, saat prospek telah berhasil dibuat yang dapat dilihat pada Gambar 3.26, maka prospek tersebut akan disimpan ke dalam *database* dan sistem akan mengirimakan *message* ke RabbitMQ. Hal ini dilakukan untuk mengurutkan prospek yang masuk dan Redis nantinya akan melakukan *caching* untuk melakukan penugasan pada agen penjualan terkait prospek tersebut.



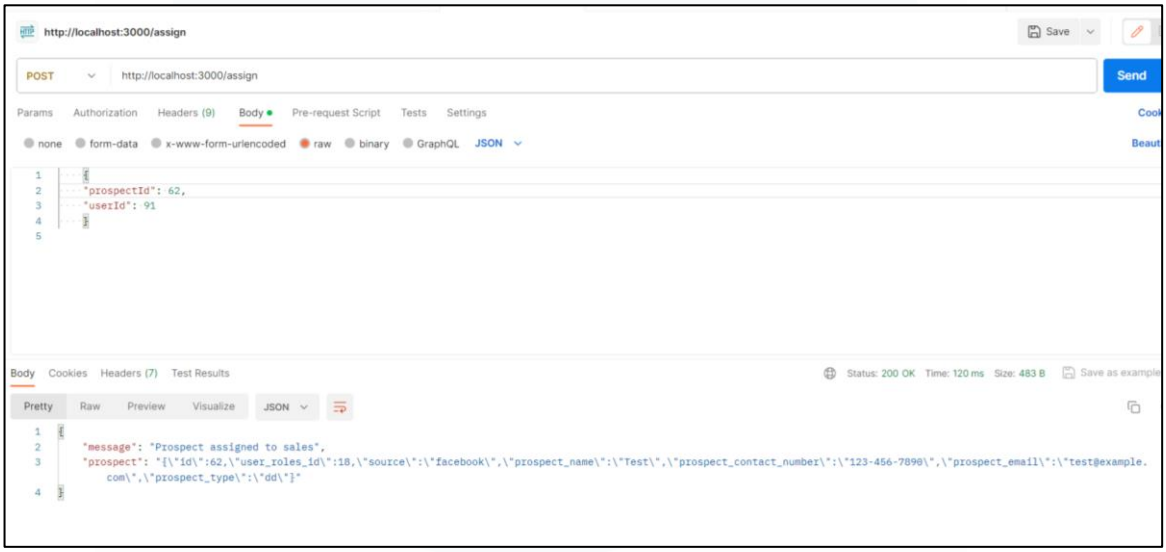
Gambar 3.26 *User* Berhasil Memasukkan Prospek di Postman

Gambar 3.26 merupakan hasil yang ditampilkan pada aplikasi Postman yang menunjukkan bahwa prospek telah berhasil disimpan ke dalam sistem.

```
app.post('/assign', async (req, res) => {
  try {
    const { prospectId, userId } = req.body;

    // Retrieve the user (sales representative) from your users table
    const user = await User.findByPk(userId);

    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }

    // Retrieve the prospect data from Redis
    const redisKey = `prospect:${prospectId}`;
    const prospectData = await redisClient.get(redisKey);

    if (!prospectData) {
      return res.status(404).json({ error: 'Prospect not found in Redis' });
    }

    // Update the prospect data in Redis
    await redisClient.setex(redisKey, 3600, JSON.stringify(prospectData));

    // Publish a message to RabbitMQ to inform about the assignment
    const rabbitMqUrl = 'amqp://localhost';
    const connection = await amqp.connect(rabbitMqUrl);
    const channel = await connection.createChannel();
    const exchangeName = 'sales-assignment';
    const message = JSON.stringify({ prospectId, userId });
    channel.assertExchange(exchangeName, 'fanout', { durable: false });
    channel.publish(exchangeName, '', Buffer.from(message));
```

Gambar 3.27 Kode pada *Assign* API

Gambar 3.27 merupakan kode penyusun pada *Assign* API. Dengan menggunakan *UserId* sebagai tanda untuk menerima prospek. Nantinya, Redis akan otomatis mengirimkan prospek kepada *User* sehingga *User* dapat langsung melakukan *follow-up* pada prospek tersebut.



Gambar 3.28 Sistem Berhasil Melakukan *Assign* secara Manual pada *User*

Gambar 3.28 merupakan hasil dari *assign* prospek kepada *User* yang dilakukan oleh sistem. Dengan hanya menggunakan *UserId* dan memilih prospek mana yang akan diberikan kepada agen penjualan untuk melanjutkan prospek.

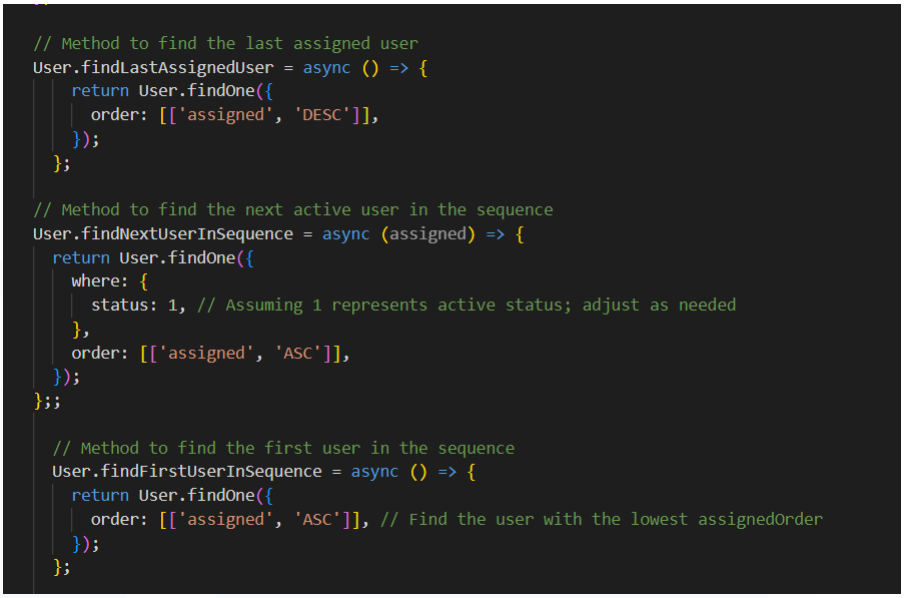
b. Merancang dan membuat *database* untuk *auto-assign* pada *website*

Sebelum membuat *auto-assign* pada prospek, maka diperlukan *database* yang mendukung hal tersebut. *Database* yang dibuat harus mendukung sistem yang ada sehingga sistem dapat berjalan dengan baik. Terdapat beberapa tabel yang dibuat di dalam *database* seperti *Users*, *prospects*, *User_roles*, *responses*. Gambar 3.22 menunjukan

relasi tabel dan *database* yang mendukung untuk melakukan *auto-assign* prospek kepada agen penjualan atau *User*.

c. Merancang dan membuat sistem *auto-assign leads* untuk *sales* penjualan.

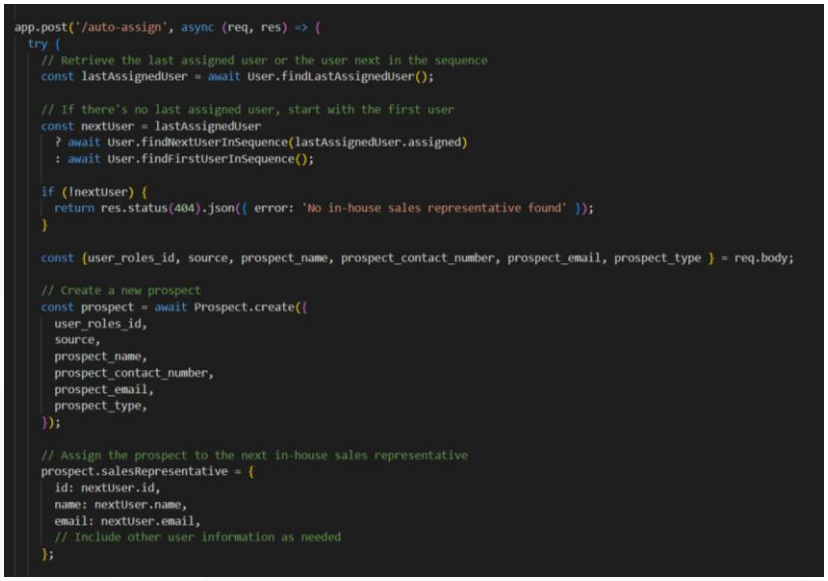
Sistem *auto-assign* yang akan dibuat untuk agen penjualan dibagi menjadi dua bagian, yaitu sesi *on-duty* dan *off-duty*. Kedua sesi ini dibuat untuk menyesuaikan jam kerja pada karyawan. Sesi *on-duty* dilakukan pada hari kerja yang dimulai dari pukul 08:00 – 18:00, para agen penjualan diberikan kebebasan untuk mengaktifkan fitur *on-duty* yang ada pada aplikasi agar dapat menerima prospek. Apabila para agen penjualan tidak menghidupkan fitur tersebut, maka para agen tidak dapat menerima prospek apapun selama masa jam kerja. Berbeda dengan *off-duty* para agen penjualan akan menerima prospek secara langsung di luar jam kerja atau setelah pukul 18:00 dan di hari libur. Gambar 3.30 menunjukkan *auto-assign* ketika para agen mengaktifkan sesi *on-duty*. Gambar 3.31 menunjukkan *auto-assign* ketika sudah memasuki sesi *off-duty*.



```
// Method to find the last assigned user
User.findLastAssignedUser = async () => {
    return User.findOne({
      order: [['assigned', 'DESC']],
    });
  };

// Method to find the next active user in the sequence
User.findNextUserInSequence = async (assigned) => {
  return User.findOne({
    where: {
      status: 1, // Assuming 1 represents active status; adjust as needed
    },
    order: [['assigned', 'ASC']],
  });
};;

  // Method to find the first user in the sequence
  User.findFirstUserInSequence = async () => {
    return User.findOne({
      order: [['assigned', 'ASC']], // Find the user with the lowest assignedOrder
    });
  };
```

Gambar 3.29 Logika yang Digunakan dalam Merancang Sistem *Auto-assign*

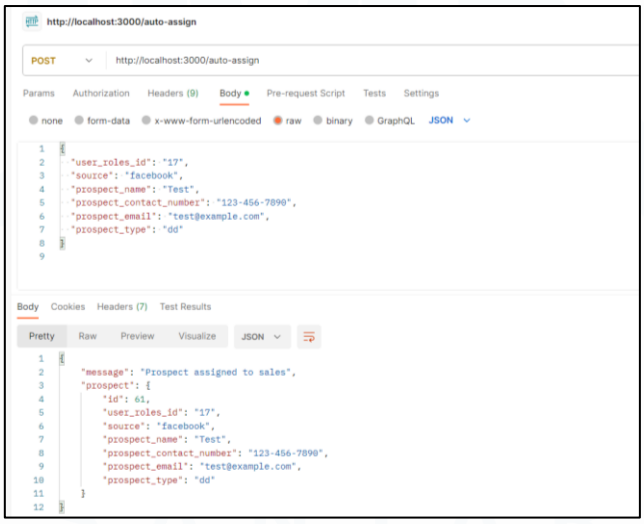
Gambar 3.29 menunjukkan beberapa logika yang digunakan dalam membuat sistem *auto-assign* berupa logika untuk mencari *User* yang terakhir kali diberikan prospek, mencari *User* selanjutnya, dan mencari *User* pertama saat sistem dijalankan. Logika ini dibuat bedasarkan jumlah prospek dan status dari masing-masing *User* dengan memanfaatkan produktivitas pada setiap *User* terhadap prospek. Untuk status disimbolkan dengan menggunakan angka 0 dan angka 1. Angka 0 menunjukkan bahwa *User* sedang tidak aktif dan angka 1 menunjukkan bahwa *User* sedang aktif dan siap untuk menerima prospek.



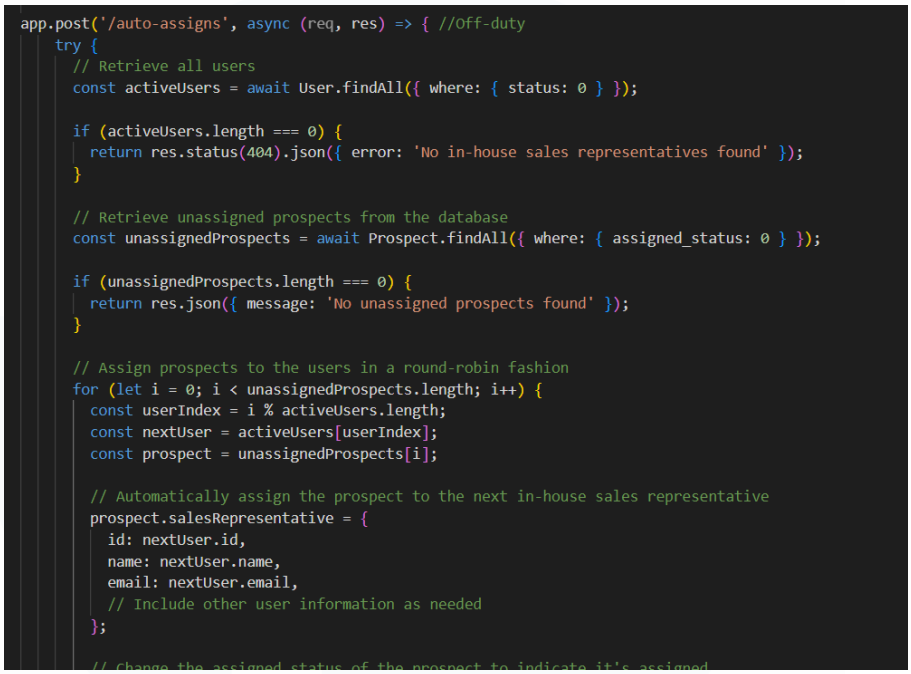
```
app.post('/auto-assign', async (req, res) => {
  try {
    // Retrieve the last assigned user or the user next in the sequence
    const lastAssignedUser = await User.findLastAssignedUser();

    // If there's no last assigned user, start with the first user
    const nextUser = lastAssignedUser
      ? await User.findNextUserInSequence(lastAssignedUser.assigned)
      : await User.findFirstUserInSequence();

    if (!nextUser) {
      return res.status(404).json({ error: 'No in-house sales representative found' });
    }

    const {user_roles_id, source, prospect_name, prospect_contact_number, prospect_email, prospect_type } = req.body;

    // Create a new prospect
    const prospect = await Prospect.create({
      user_roles_id,
      source,
      prospect_name,
      prospect_contact_number,
      prospect_email,
      prospect_type,
    });

    // Assign the prospect to the next in-house sales representative
    prospect.salesRepresentative = {
      id: nextUser.id,
      name: nextUser.name,
      email: nextUser.email,
      // Include other user information as needed
    };
```

Gambar 3.30 Kode pada *Auto-assign* API ketika *User* Aktif

Gambar 3.30 merupakan tampilan kode pada *Auto-assign* API ketika *User* sedang aktif atau *on-duty*. Di dalam kode tersebut tidak memiliki metode pengulangan dikarenakan sistem akan mengecek terlebih dahulu apakah terdapat *User* yang sedang aktif atau tidak sebelum menjalankan kode selanjutnya. Gambar 3.31 merupakan tampilan pada aplikasi Postman yang menunjukkan bahwa sistem berhasil melakukan *auto-assign*.



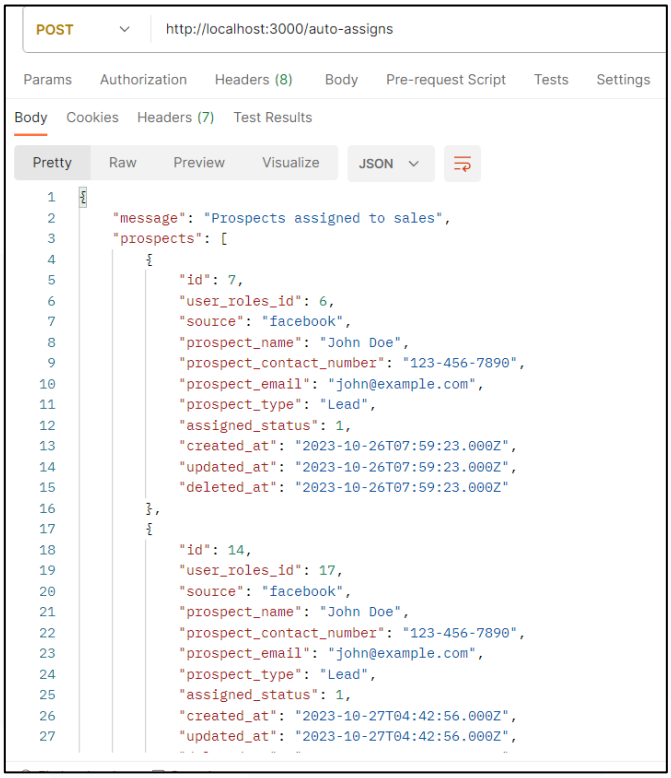
Gambar 3.31 Sistem Berhasil Melakukan *Auto-assign* Ketika *User* Aktif



```
app.post('/auto-assigns', async (req, res) => { //Off-duty
    try {
      // Retrieve all users
      const activeUsers = await User.findAll({ where: { status: 0 } });

      if (activeUsers.length === 0) {
        return res.status(404).json({ error: 'No in-house sales representatives found' });
      }

      // Retrieve unassigned prospects from the database
      const unassignedProspects = await Prospect.findAll({ where: { assigned_status: 0 } });

      if (unassignedProspects.length === 0) {
        return res.json({ message: 'No unassigned prospects found' });
      }

      // Assign prospects to the users in a round-robin fashion
      for (let i = 0; i < unassignedProspects.length; i++) {
        const userIndex = i % activeUsers.length;
        const nextUser = activeUsers[userIndex];
        const prospect = unassignedProspects[i];

        // Automatically assign the prospect to the next in-house sales representative
        prospect.salesRepresentative = {
          id: nextUser.id,
          name: nextUser.name,
          email: nextUser.email,
          // Include other user information as needed
        };

        // Change the assigned status of the prospect to indicate it's assigned
```

Gambar 3.32 Kode pada *Auto-assign* API ketika *User* Tidak Aktif

Gambar 3.32 merupakan tampilan kode pada *Auto-assign* API ketika *User* tidak aktif sehingga sistem akan otomatis memberikan prospek kepada agen penjualan. Di dalam kode terdapat metode pengulangan untuk memastikan bahwa prospek yang ada terkirim pada agen penjualan. Gambar 3.33 merupakan tampilan sistem yang berhasil menjalankan *auto-assign* di *Postman*.



Gambar 3.33 Sistem Berhasil Melakukan *Auto-assign* Ketika *User* Tidak Aktif

Setelah membuat *auto-assign* API dimana prospek terdaftar di luar jam kerja dan sudah tersimpan ke dalam *database*, maka dilanjutkan dengan pembuatan *auto-assign* API ketika prospek baru masuk saat agen penjualan sedang melakukan *follow-up* pada prospek lain. Pada sistem ini prospek yang baru masuk akan tersimpan ke dalam *database* dan akan langsung terkirim pada *User* atau agen penjualan yang sedang aktif. Nantinya, *User* atau agen penjualan akan merespon prospek tersebut apakah diterima atau ditolak dalam rentang waktu tertentu. Apabila prospek diterima, maka prospek akan diberikan kepada *User* tersebut. Apabila prospek ditolak atau prospek tidak direspon oleh *User* maka secara otomatis sistem akan mengirimkan prospek tersebut pada *User* lain yang sedang aktif. Gambar 3.34 merupakan tampilan kode pada *auto-assign-User* API dan Gambar 3.36 merupakan tampilan kode untuk mengatasi permasalahan pada *auto-assign-User* API apabila terjadi penolakan atau tidak ada respon. Sistem

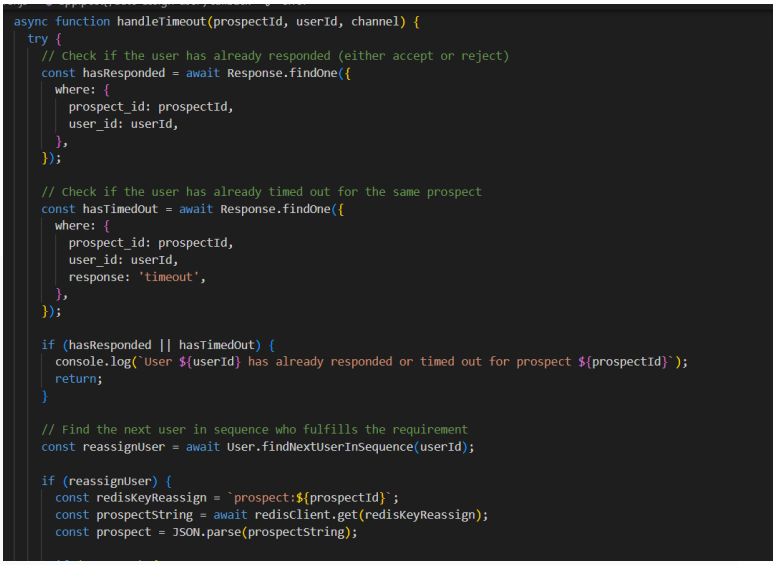
juga akan mencatat respon dari para *User* ke dalam *database* yang terlihat pada Gambar 3.35.

```
const prospect = await Prospect.create({
  user_roles_id,
  source,
  prospect_name,
  prospect_contact_number,
  prospect_email,
  prospect_type,
});

prospect.salesRepresentative = {
  id: nextUser.id,
  name: nextUser.name,
  email: nextUser.email,
};

await prospect.save();

const redisKey = `prospect:${prospect.id}`;
const redisValue = JSON.stringify(prospect);
redisClient.setex(redisKey, 3600, redisValue);

const rabbitMqUrl = 'amqp://localhost';
const connection = await amqp.connect(rabbitMqUrl);
const channel = await connection.createChannel();
const exchangeName = 'sales-assignment';
const message = JSON.stringify({ prospectId: prospect.id, userId: nextUser.id });
channel.assertExchange(exchangeName, 'fanout', { durable: false });
channel.publish(exchangeName, '', Buffer.from(message));

const timer = setTimeout(async () => {
  await handleTimeout(prospect.id, nextUser.id, channel);
}, 180000); // 3 minutes
```

Gambar 3.34 Kode pada *Auto-assign* ketika Prospek Baru Masuk

```
// Update the prospect's assigned status based on the user's response
if (response === 'accept') {
  if (prospect.assigned === 1) {
    // If already accepted, return an error
    return res.status(400).json({ error: 'User has already accepted the prospect' });
  }

  prospect.assigned = 1; // Assuming 1 represents accepted status; adjust as needed

  // Update the AssignedProspect table only if the user accepted
  const assignmentDate = new Date();
  await AssignedProspect.create({
    prospect_id: prospectId,
    user_id: assignedUserId,
    assignment_date: assignmentDate,
  });

  // Update the user's assigned count
  const assignedUserModel = await User.findByPk(assignedUserId);
  if (assignedUserModel instanceof User) {
    assignedUserModel.assigned += 1;
    await assignedUserModel.save();
  }

  // Update the prospect data in Redis
  const redisValue = JSON.stringify(prospect);
  await redisClient.setex(redisKey, 3600, redisValue); // Use await here

  // Update the assigned_status in the Prospect table to 1
  await Prospect.update({ assigned_status: 1 }, { where: { id: prospectId } });
}
```

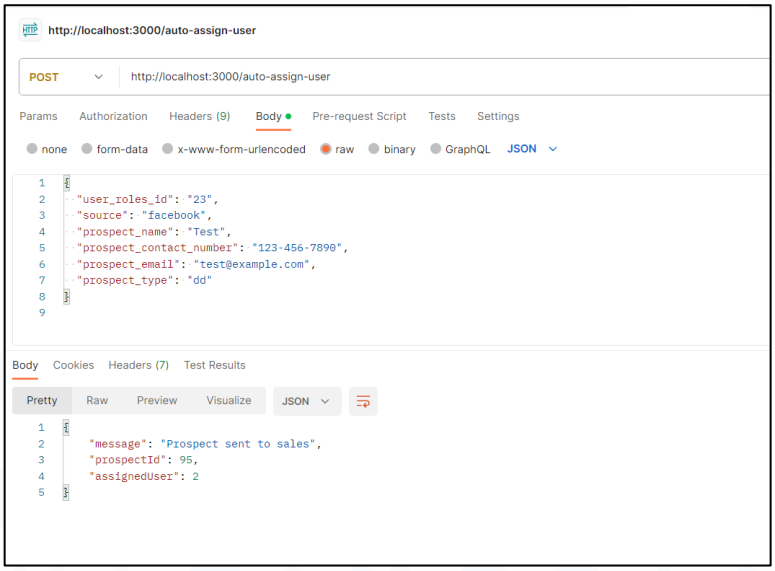
Gambar 3.35 Kode pada *User-response* ketika *User* Merespon Prospek



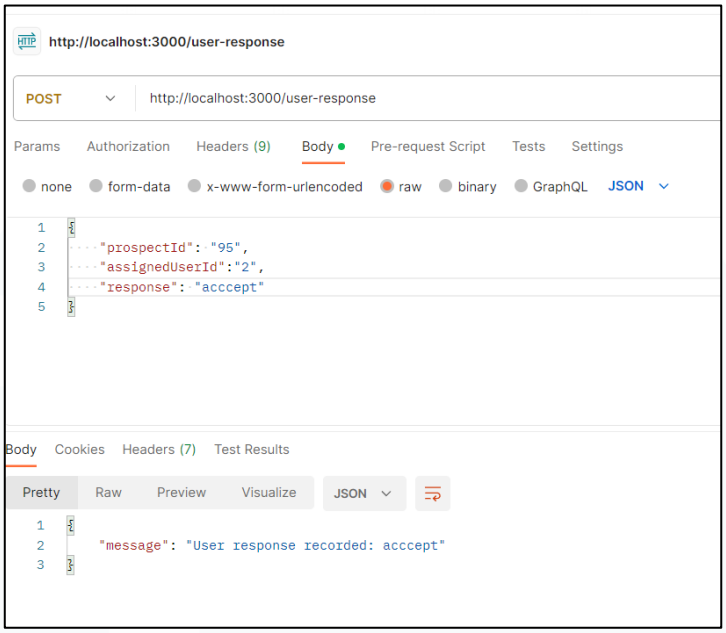
```
async function handleTimeout(prospectId, userId, channel) {
  try {
    // Check if the user has already responded (either accept or reject)
    const hasResponded = await Response.findOne({
      where: {
        prospect_id: prospectId,
        user_id: userId,
      },
    });

    // Check if the user has already timed out for the same prospect
    const hasTimedOut = await Response.findOne({
      where: {
        prospect_id: prospectId,
        user_id: userId,
        response: 'timeout',
      },
    });

    if (hasResponded || hasTimedOut) {
      console.log(`User ${userId} has already responded or timed out for prospect ${prospectId}`);
      return;
    }

    // Find the next user in sequence who fulfills the requirement
    const reassignUser = await User.findNextUserInSequence(userId);

    if (reassignUser) {
      const redisKeyReassign = `prospect:${prospectId}`;
      const prospectString = await redisClient.get(redisKeyReassign);
      const prospect = JSON.parse(prospectString);
```

Gambar 3.36 Kode untuk Mengatasi Permasalahan pada *Auto-assign* API

Gambar 3.36 merupakan kode untuk mengatasi permasalahan yang muncul ketika *User* melakukan *reject* pada prospek atau tidak merespon pada prospek yang diberikan kepada *User* tersebut.



Gambar 3.37 Sistem Berhasil Mengirimkan Prospek kepada *User*



Gambar 3.38 Sistem Mencatat Respon *User*

Gambar 3.38 merupakan tampilan pada aplikasi Postman yang menunjukkan bahwa sistem berhasil mencatat respon dari *User* terkait dengan prospek yang diberikan.

d. Membuat *unit-testing* untuk mengecek kode yang telah dibuat

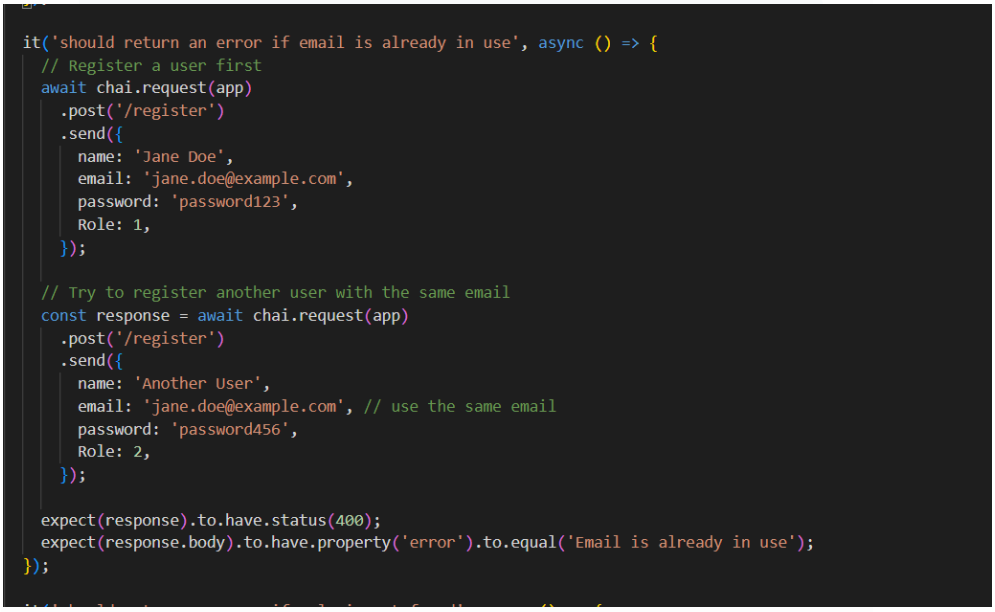
adalah pengujian yang dilakukan hingga tingkat terkecil pada suatu sistem untuk memastikan bahwa setiap unit kode berfungsi dengan benar sesuai dengan spesifikasi yang diinginkan. dalam penerapannya dapat mendeteksi kesalahan sejak awal sehingga sangat membantu dalam pengembangan suatu sistem. dalam penerapannya juga dapat dilakukan secara otomatis sehingga pengujian dapat lebih konsisten dan efisien. Secara tidak langsung, *unit testing* juga dapat meningkatkan kemanan pada sistem karena dapat mengidentifikasi *bug* yang ada pada sistem lebih awal yang dapat mengurangi potensi serangan keamanan pada sistem. Oleh karena itu, dapat diketahui bahwa *unit testing* bukan hanya sekedar sebagai alat pengujian saja, tetapi juga berperan penting dalam pengembangan sistem sesuai dengan kebutuhan yang diingikan.

Tahap awal yang dilakukan dalam pembuatan secara otomatis adalah dengan menentukan setiap kasus atau permasalahan yang mungkin muncul ketika sistem dijalankan. Setelah mengidentifikasi kemungkinan yang ada, maka pembuatan kode untuk *unit testing* dapat dimulai. Pembuatan dilakukan dengan menggunakan *library chai* dan *mocha* yang biasa digunakan pada *node.js*. Kedua *library* ini biasa digunakan secara bersamaan dalam pembuatan *unit testing* dikarenakan adanya fungsi yang saling berhubungan. *Library mocha* berfungsi untuk menyediakan struktur dalam pengujian seperti *describe* dan *it* serta memungkinkan untuk mengatur siklus hidup pengujian. *Library chai* berfungsi untuk melakukan pengujian dengan gaya asertif seperti *should, expect,* dan *assert* serta memungkinkan untuk pengecekan berbagai tipe dan struktur data yang hasilnya dapat mudah dipahami. Dengan melakukan kombinasi antara kedua *library* ini maka hasil pengujian dapat dilakukan secara maksimal. Setelah berhasil melakukan maka sistem akan menyatakan berjalan atau tidak berjalan kode tersebut. Apabila sistem menyatakan bahwa adanya kegagalan atau tidak berjalannya kode, maka terdapat kesalahan penulisan kode pada *file* utama sehingga harus diperbaiki.

```
describe('POST /register', () => {
  after(async () => {
  });

  it('should successfully register a user', async () => {
    const response = await chai.request(app)
      .post('/register')
      .send({
        name: 'John Doe',
        email: 'john.doe@example.com',
        password: 'password123',
        Role: 6,
      });

    expect(response).to.have.status(201);
    expect(response.body).to.have.property('message').to.equal('User registered');
    expect(response.body).to.have.property('user');
    expect(response.body.user).to.have.property('name').to.equal('John Doe');
    // Add more assertions based on the expected structure of the response
  });
```

Gambar 3.39 Kode *Unit Testing* saat *User* Melakukan Registrasi

Gambar 3.39 merupakan kode untuk melakukan pengetesan pada *registration* API untuk memastikan bahwa registrasi pada *User* telah berhasil dilakukan dan data yang ada sudah sesuai dengan data saat registrasi dilakukan.



```
it('should return an error if email is already in use', async () => {
  // Register a user first
  await chai.request(app)
    .post('/register')
    .send({
      name: 'Jane Doe',
      email: 'jane.doe@example.com',
      password: 'password123',
      Role: 1,
    });

  // Try to register another user with the same email
  const response = await chai.request(app)
    .post('/register')
    .send({
      name: 'Another User',
      email: 'jane.doe@example.com', // use the same email
      password: 'password456',
      Role: 2,
    });

  expect(response).to.have.status(400);
  expect(response.body).to.have.property('error').to.equal('Email is already in use');
});
```

Gambar 3.40 Kode *Unit Testing* untuk Mengecek Duplikasi E-mail *User*

Gambar 3.40 merupakan kode untuk melakukan pengetesan pada *registration* API untuk memastikan tidak adanya duplikasi e-mail saat melakukan registrasi. Hal ini dilakukan untuk mencegah adanya *User* yang menggunakan e-mail yang sama untuk membuat akun yang sama.

```
  it('should return an error if role is not found', async () => {
    const response = await chai.request(app)
      .post('/register')
      .send({
        name: 'Invalid Role User',
        photo: 'tes.jpg',
        username: 'tes',
        email: 'invalid.role@example.com',
        password: 'password789',
        Role: 10,
      });

    expect(response).to.have.status(400);
    expect(response.body).to.have.property('error').to.equal('Role not found');
  });
});
```

Gambar 3.41 Kode *Unit Testing* ketika *Role* Tidak Ditemukan

Gambar 3.41 merupakan kode untuk melakukan pengetesan untuk memastikan *role* yang diisi oleh *User* sudah sesuai dengan ketentuan yang ada. Apabila *role* yang diisi oleh *User* tidak sesuai dengan sistem, maka registrasi akan dibatalkan secara otomatis.

```
describe('POST /login', () => {
  after(async () => {
  });

  it('should successfully log in a user', async () => {
    // Register a user first
    await chai.request(app)
      .post('/register')
      .send({
        name: 'Login User',
        email: 'login.user@example.com',
        password: 'password123',
        Role: 6,
      });

    // Log in the user
    const response = await chai.request(app)
      .post('/login')
      .send({
        email: 'login.user@example.com',
        password: 'password123',
      });

    expect(response).to.have.status(200);
    expect(response.body).to.have.property('token');
    // Add more assertions based on the expected structure of the response
  });
```

Gambar 3.42 Kode *Unit Testing* saat *User* Login

Gambar 3.42 merupakan kode untuk melakukan pengetesan pada *User* yang ingin melakukan *login*. Setelah sistem sudah memastikan bahwa *User* sudah melakukan *login*, maka sistem juga akan mengecek bahwa token JWT yang diberikan kepada *User* sudah diberikan.

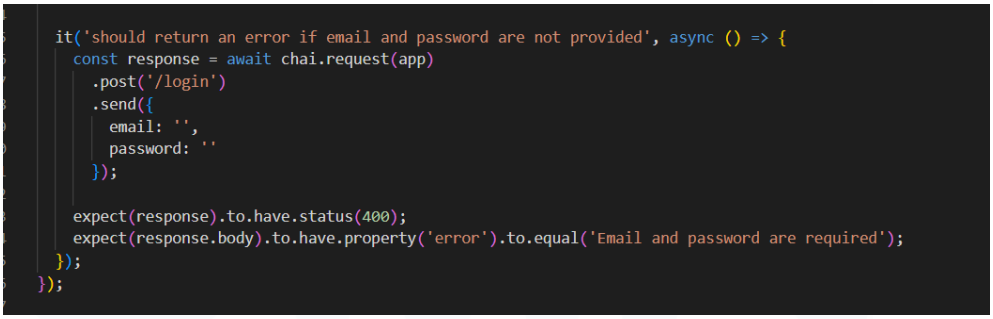
```
it('should return an error if email is invalid', async () => {
  const response = await chai.request(app)
    .post('/login')
    .send({
      email: 'nonexistent.user@example.com', // use a nonexistent email
      password: 'password456',
    });

  expect(response).to.have.status(401);
  expect(response.body).to.have.property('error').to.equal('Invalid email');
});
```

Gambar 3.43 Kode *Unit Testing* saat E-mail Tidak Valid

Gambar 3.43 merupakan kode untuk melakukan pengecekan bahwa e-mail yang digunakan oleh *User* sudah sesuai dengan data yang ada. Apabila e-mail tidak sesuai dengan sistem, maka *login* akan otomatis dibatalkan.

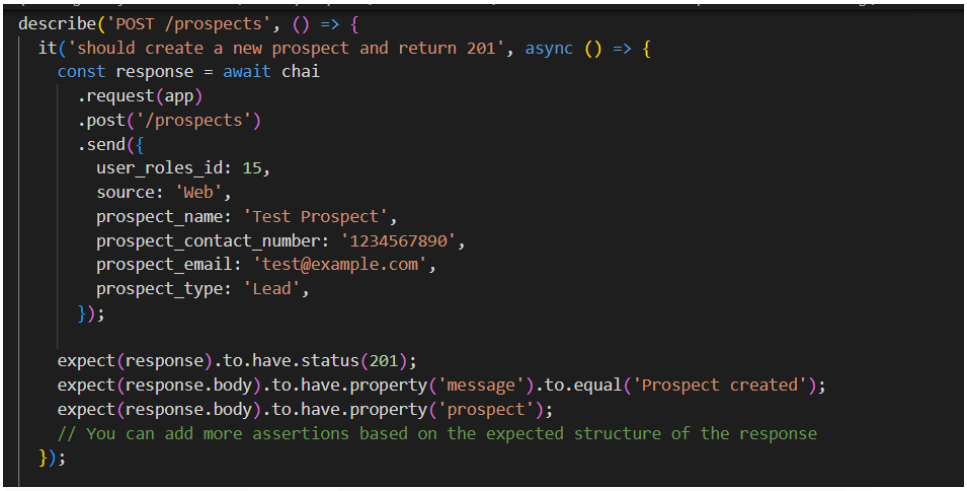
Gambar 3.44 Kode *Unit Testing* saat *Password* Salah

Gambar 3.44 merupakan kode untuk melakukan pengecekan bahwa *password User* sudah benar, apabila *password* yang dimasukkan oleh *User* gagal maka sistem kana otomatis membatalkan proses *login*.



Gambar 3.45 Kode *Unit Testing* saat E-mail dan *Password* Kosong

Gambar 3.45 merupakan kode untuk melakukan pengecekan bahwa e-mail dan *password* sudah diisi oleh *User*. Apabila kedua data tersebut kosong, maka sistem akan membatalkan proses *login*.



```
describe('POST /prospects', () => {
  it('should create a new prospect and return 201', async () => {
    const response = await chai
      .request(app)
      .post('/prospects')
      .send({
        user_roles_id: 15,
        source: 'Web',
        prospect_name: 'Test Prospect',
        prospect_contact_number: '1234567890',
        prospect_email: 'test@example.com',
        prospect_type: 'Lead',
      });

    expect(response).to.have.status(201);
    expect(response.body).to.have.property('message').to.equal('Prospect created');
    expect(response.body).to.have.property('prospect');
    // You can add more assertions based on the expected structure of the response
  });
```

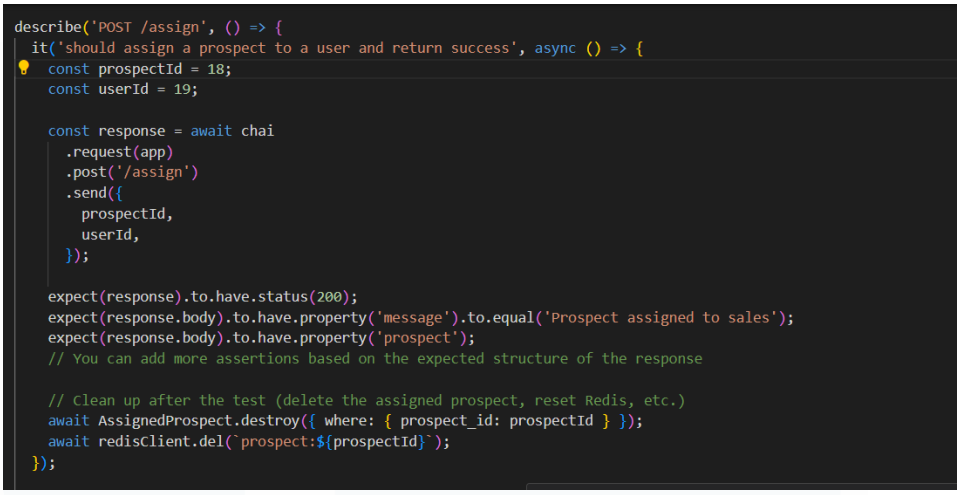
Gambar 3.46 Kode untuk *Prospect* API

Gambar 3.46 merupakan kode untuk melakukan pengetesan pada *prospect* API. Dengan mengisi data yang telah ditentukan oleh sistem, maka prospek dapat dengan mudah tercatat dalam sistem. Dalam mengisi prospek *User* juga harus mengisi *User_roles_id* untuk mengetahui darimana prospek tersebut berasal dari *User* mana. Tujuan dilakukan hal ini adalah untuk mengetahui peforma dari setiap *User* dan untuk dilakukan analisis nantinya.

```
it('should return an error if required fields are missing', async () => {
  const response = await chai.request(app).post('/prospects').send({

  });

  expect(response).to.have.status(400);
  expect(response.body).to.have.property('error');
});

it('should handle database errors and return 500', async () => {
  const response = await chai.request(app).post('/prospects').send({
    user_roles_id: 999, // Non-existent user_roles_id
    source: 'Web',
    prospect_name: 'Test Prospect',
    prospect_contact_number: '1234567890',
    prospect_email: 'test@example.com',
    prospect_type: 'Lead',
  });

  expect(response).to.have.status(500);
  expect(response.body).to.have.property('error');
  // You can add more assertions based on the expected structure of the response
});
```

Gambar 3.47 Kode *Unit Testing* untuk Mengecek *Error*

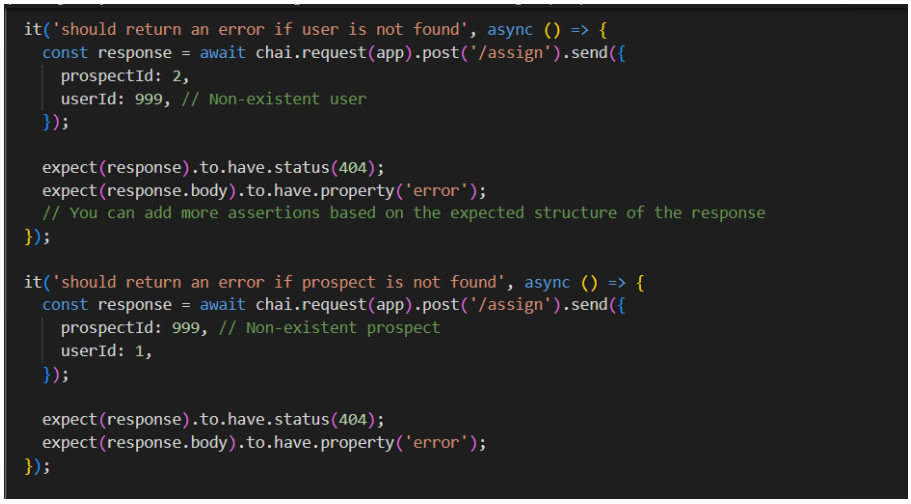
Gambar 3.47 merupakan kode untuk mengecek dan mengatasi *error* ketika *User* ingin memasukkan prospek ke dalam sistem. Dalam hal ini propek tidak akan terdaftar apabila *User* tidak mencantumkan *User_roles_id* dan tidak mengisi data prospek secara keseluruhan. Namun, apabila hanya beberapa data yang tidak diisi sistem tetap menerima data prospek tersebut.



```
describe('POST /assign', () => {
  it('should assign a prospect to a user and return success', async () => {
    const prospectId = 18;
    const userId = 19;

    const response = await chai
      .request(app)
      .post('/assign')
      .send({
        prospectId,
        userId,
      });

    expect(response).to.have.status(200);
    expect(response.body).to.have.property('message').to.equal('Prospect assigned to sales');
    expect(response.body).to.have.property('prospect');
    // You can add more assertions based on the expected structure of the response

    // Clean up after the test (delete the assigned prospect, reset Redis, etc.)
    await AssignedProspect.destroy({ where: { prospect_id: prospectId } });
    await redisClient.del(`prospect:${prospectId}`);
  });
```

Gambar 3.48 Kode *Unit Testing* untuk *Assign* API

Gambar 3.48 merupakan kode untuk melakukan pengetesan pada *assign* API. Sistem akan memberikan prospek kepada *User* secara manual dengan menggunakan *prospectId* dan *UserId*. Nantinya, akan dilakukan pengecekan apakah prospek sudah benar-benar terkirim kepada *User* melalui pengecekan ini.



```
  it('should return an error if user is not found', async () => {
    const response = await chai.request(app).post('/assign').send({
      prospectId: 2,
      userId: 999, // Non-existent user
    });

    expect(response).to.have.status(404);
    expect(response.body).to.have.property('error');
    // You can add more assertions based on the expected structure of the response
  });

  it('should return an error if prospect is not found', async () => {
    const response = await chai.request(app).post('/assign').send({
      prospectId: 999, // Non-existent prospect
      userId: 1,
    });

    expect(response).to.have.status(404);
    expect(response.body).to.have.property('error');
  });
```

Gambar 3.49 Kode *Unit Testing* untuk Mengatasi *Error*

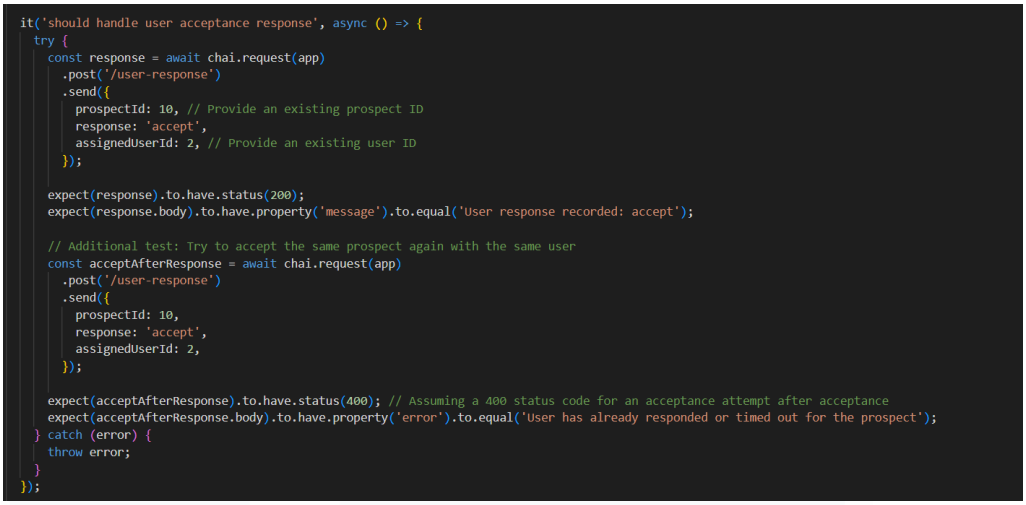
Gambar 3.49 merupakan kode untuk melakukan pengecekan apabila terjadi *error* pada saat dilakukan *assign* kepada *User*. Pengecekan ini

dilakukan dengan memastikan bahwa terdapat *prospek* dan *User* yang tersimpan di dalam *database* untuk mengurangi kesalahan ataupun duplikasi data saat melakukan *assign*.

```
it('should auto-assign a prospect to a user', async () => {
  try {
    const response = await chai.request(app)
      .post('/auto-assign-user')
      .send({
        user_roles_id: 23,
        source: 'Test Source',
        prospect_name: 'Test Prospect',
        prospect_contact_number: '1234567890',
        prospect_email: 'test@example.com',
        prospect_type: 'Test Type',
      });

    expect(response).to.have.status(200);
    expect(response.body).to.have.property('message').to.equal('Prospect sent to sales');
    expect(response.body).to.have.property('prospectId').to.be.a('number');
    expect(response.body).to.have.property('assignedUser').to.be.a('number');
  } catch (error) {
    throw error;
  }
});
```

Gambar 3.50 Kode *Unit Testing* untuk *Auto-assign*

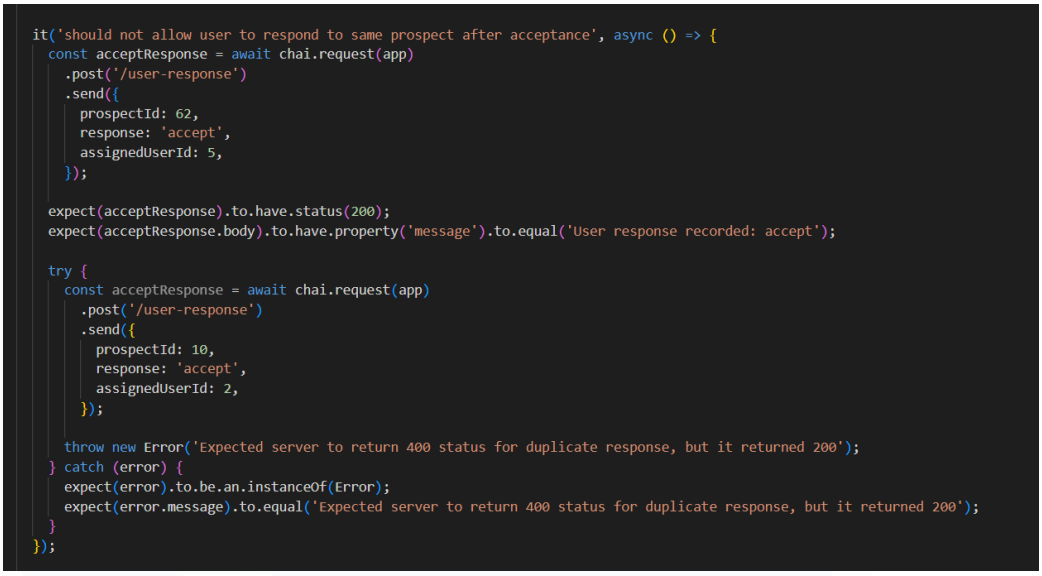
Gambar 3.50 merupakan kode untuk melakukan pengecekan apakah prospek sudah terkirim secara otomatis kepada *User* yang sedang aktif ketika prospek baru saja diterima. Untuk memastikan bahwa prospek sesuai dengan *User* maka setiap *id* pada prospek haruslah sesuai dengan ketika saat *User* menerima prospek tersebut.



```
it('should handle user acceptance response', async () => {
  try {
    const response = await chai.request(app)
      .post('/user-response')
      .send({
        prospectId: 10, // Provide an existing prospect ID
        response: 'accept',
        assignedUserId: 2, // Provide an existing user ID
      });

    expect(response).to.have.status(200);
    expect(response.body).to.have.property('message').to.equal('User response recorded: accept');

    // Additional test: Try to accept the same prospect again with the same user
    const acceptAfterResponse = await chai.request(app)
      .post('/user-response')
      .send({
        prospectId: 10,
        response: 'accept',
        assignedUserId: 2,
      });

    expect(acceptAfterResponse).to.have.status(400); // Assuming a 400 status code for an acceptance attempt after acceptance
    expect(acceptAfterResponse.body).to.have.property('error').to.equal('User has already responded or timed out for the prospect');
  } catch (error) {
    throw error;
  }
});
```

Gambar 3.51 Kode *Unit Testing* untuk *User*-response

Gambar 3.51 merupakan kode untuk mengecek apakah sistem sudah mencatat apapun respon dari *User*, termasuk penolakan atau tidak ada respon. Kode ini juga memastikan bahwa respon yang diberikan oleh *User* pada setiap prospeknya hanya boleh satu kali saja sehingga jika *User* mengirimkan respon lebih dari satu kali, sistem akan secara otomatis akan menolak respon tersebut dan memberitahu *User* bahwa *User* sudah merespon prospek tersebut.



```
it('should not allow user to respond to same prospect after acceptance', async () => {
  const acceptResponse = await chai.request(app)
    .post('/user-response')
    .send({
      prospectId: 62,
      response: 'accept',
      assignedUserId: 5,
    });

  expect(acceptResponse).to.have.status(200);
  expect(acceptResponse.body).to.have.property('message').to.equal('User response recorded: accept');

  try {
    const acceptResponse = await chai.request(app)
      .post('/user-response')
      .send({
        prospectId: 10,
        response: 'accept',
        assignedUserId: 2,
      });

    throw new Error('Expected server to return 400 status for duplicate response, but it returned 200');
  } catch (error) {
    expect(error).to.be.an.instanceOf(Error);
    expect(error.message).to.equal('Expected server to return 400 status for duplicate response, but it returned 200');
  }
});
```

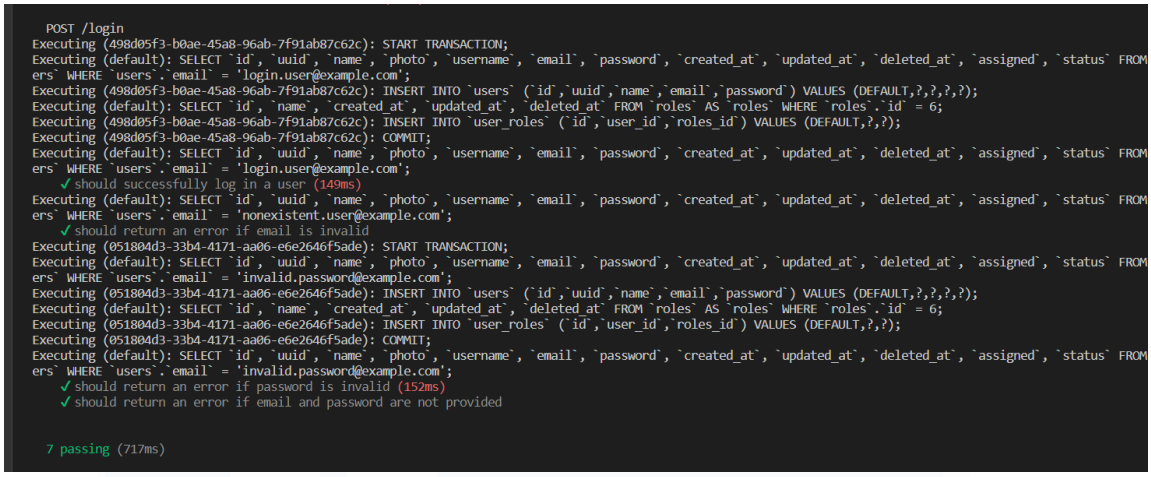
Gambar 3.52 Kode *Unit Testing* untuk Mengatasi Masalah Respon *User*

Gambar 3.52 merupakan kode untuk memastikan *User* untuk tidak merespon lagi pada prospek yang sudah diterima oleh *User*. Hal ini dilakukan untuk mencegah adanya redudansi data pada *database* dan kesalahan penghitungan pada prospek yang telah dikerjakan oleh *User*. Total perhitungan pada prospek sangatlah penting bagi *User* untuk mengetahui peforma dari *User* itu sendiri.

```
it('should handle user not responding and reassign to another user after timeout', async () => {
  try {
    const response = await chai.request(app)
      .post('/user-response')
      .send({
        prospectId: 10, // Provide an existing prospect ID
        response: 'timeout', // User does not respond
        assignedUserId: 2, // Provide an existing user ID
      });

    expect(response).to.have.status(200);
    expect(response.body).to.have.property('message').to.equal('User response recorded: timeout');

    // Add assertions for stubbed services if needed

    // Advance the clock to simulate a timeout
    clock.tick(180001); // 3 minutes and 1 millisecond

    // The prospect should be automatically reassigned to another user
    const reassignResponse = await chai.request(app)
      .post('/auto-assign-user')
      .send({
        user_roles_id: 23,
        source: 'Test Source',
        prospect_name: 'Test Prospect',
        prospect_contact_number: '1234567890',
        prospect_email: 'test@example.com',
        prospect_type: 'Test Type',
      });

    expect(reassignResponse).to.have.status(200);
    expect(reassignResponse.body).to.have.property('message').to.equal('Prospect sent to sales');
    expect(reassignResponse.body).to.have.property('prospectId').to.be.a('number');
    expect(reassignResponse.body).to.have.property('assignedUser').to.be.a('number').to.not.equal(2);
```

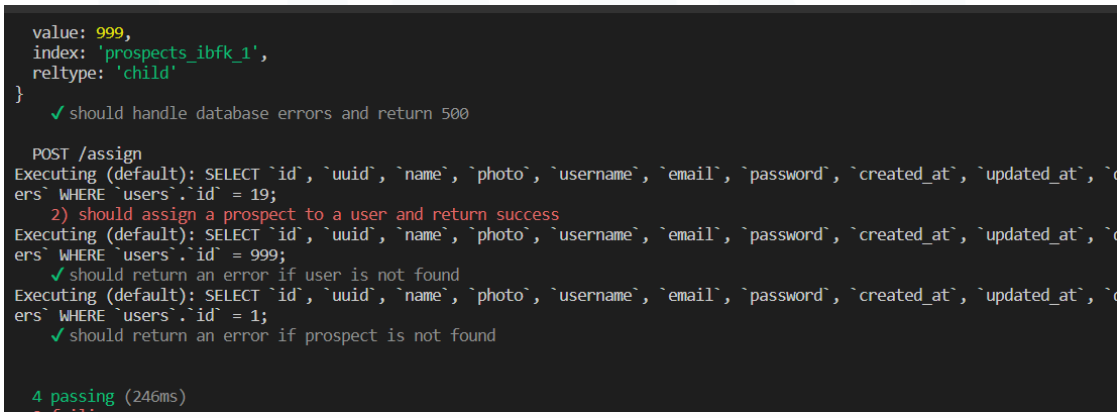
Gambar 3.53 Kode *Unit Testing* untuk Melakukan *Assign* Ulang

Gambar 3.53 kode ini juga terdapat pengetesan untuk memastikan bahwa sistem akan mengirimkan prospek ulang ketika *User* melakukan *reject* terhadap prospek atau tidak memberikan respon pada prospek. Pada pengetesan sendiri akan dilakukan pengecekan apakah *User* yang menerima prospek adalah *User* yang berbeda dari sebelumnya dan sistem berjalan secara otomatis apabila salah satu syarat tersebut terpenuhi.



Gambar 3.54 Hasil *Unit Testing* pada *Registration-login* API

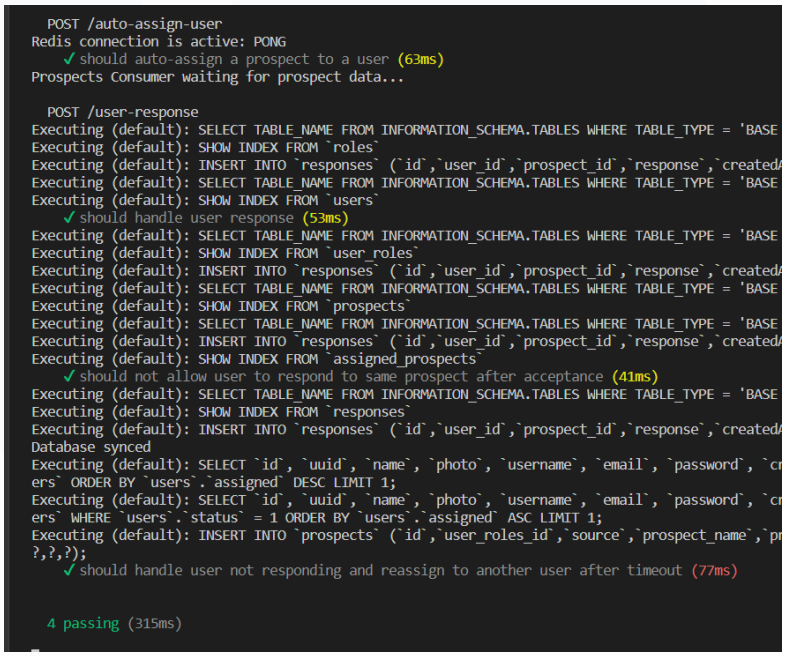
Gambar 3.54 merupakan hasil dari *unit testing* yang dilakukan pada *registration* dan *login* API yang berjumlah 7 kasus kemungkinan yang dapat terjadi pada API tersebut. Fokus utama dalam pembuatan *unit testing* pada bagian ini adalah dengan memastikan bahwa data tersimpan dengan baik pada *database* dan kode telah berjalan sesuai dengan keinginan seperti mencegah adanya e-mail yang sama sehingga *User* hanya memiliki satu akun saja.



Gambar 3.55 Hasil *Unit Testing* pada *Prospect-assign* API

Gambar 3.55 merupakan hasil dari *unit testing* yang dilakukan pada *prospect* dan *assign* API yang berjumlah 4 kasus kemungkinan yang dapat

terjadi pada API tersebut. Fokus utama dalam pembuatan *unit testing* pada bagian ini adalah memastikan prospek tercatat dan *assign* dapat dilakukan secara manual.



Gambar 3.56 Hasil *Unit Testing* pada *Auto-assign*

Gambar 3.55 merupakan hasil *unit testing* yang dilakukan pada *Auto-assign* API dan *User-response* API yang berjumlah 4 kasus kemungkinan yang dapat terjadi pada kedua API tersebut. Kedua API ini sengaja digabungkan untuk dilakukan tes secara bersamaan dikarenakan saling berhubungan satu sama lain. Fokus utama dalam pembuatan ini adalah memastikan bahwa prospek yang dikirimkan kepada agen penjualan berjalan secara otomatis dan apabila terdapat *User* yang tidak merespon atau menolak prospek maka sistem juga secara otomatis akan memberikan prospek kepada agen penjualan lain yang sedang aktif. Dalam menjalankan sistem tersebut, *User-response* API berperan untuk melakukan pencatatan respon pada setiap *User* pada setiap prospek. Untuk merangkum seluruh data, maka *User* yang menerima prospek akan tercatat dalam tabel baru sehingga mempermudah analisis nantinya.

3.4 Kendala yang Ditemukan

Dalam menjalankan program magang tentunya tidak selalu berjalan dengan lancar dan pastinya akan bertemu dengan masalah yang muncul pada saat pelaksanaan kerja. Berikut terdapat beberapa kendala yang dihadapi selama pelaksanaan program magang di PT Quanta Land Indonesia:

1. Terdapat beberapa *framework* seperti RabbitMQ dan Redis yang belum pernah dipelajari sehingga diperlukan waktu untuk mempelajari *framework* yang akan digunakan.
2. Terdapat *error* atau *bug* yang terlewatkan saat melakukan *testing* secara manual.

3.5 Solusi atas Kendala yang Ditemukan

Setiap kendala atau permasalahan yang ditemukan dalam proses pelaksanaan magang, tentunya juga terdapat solusi untuk menyelesaikan kendala tersebut. Berikut terdapat beberapa solusi yang digunakan selama pelaksanaan magang di PT Quanta Land Indonesia:

1. Belajar mandiri pada *framework* yang belum pernah dipelajari melalui video tutorial yang ada di internet untuk meningkatkan pengetahuan terhadap *framework* tersebut.
2. Membahas permasalahan atau kendala yang ditemukan ketika mengikuti *Daily Meeting* sehingga dapat dibahas secara bersama dengan tim.